

TECHNICAL DOCUMENT 3098
January 2000

**Video Imaging System (VIS)
Interface Board for the
Mobile Inshore Undersea
Warfare (MIUW) System**

J. Coleman
W. Marsh

Approved for public release;
distribution is unlimited.



20000425 126

SSC San Diego
San Diego, CA 92152-5001

SSC SAN DIEGO
San Diego, California 92152-5001

E. L. Valdes, CAPT, USN
Commanding Officer

R. C. Kolb
Executive Director

ADMINISTRATIVE INFORMATION

The work described in this document was performed for Space and Naval Warfare Systems Command by the Applied Systems Branch (D372), SPAWAR Systems Center, San Diego (SSC San Diego).

Released by
W. Marsh, Head
Applied Systems Branch

Under authority of
C. D. Metz, Head
Advanced Systems
Division

This technology is related to the subject of one or more invention disclosures assignable to the U. S. Government including N.C. 79889.

Licensing inquiries may be directed to:

Harvey Fendelman
Legal Counsel for Patents
Space and Naval Warfare Systems Center, D0012
San Diego, CA 92152-5765
(619) 553-3001

SB

CONTENTS

BACKGROUND	1
BOARD REQUIREMENTS	1
VIDEO IMAGING SYSTEM (VIS) INTERFACE BOARD	1
INTERFACE REQUIREMENTS	2
HARDWARE	2
SOFTWARE	5
HARDWARE DESIGN	7
FLASH REPROGRAMMING	9
PAL INPUTS	9
PAL OUTPUTS	9
FLASH REPROGRAMMING PROCEDURE	10
SOFTWARE DESIGN	11
SOFTWARE DEVELOPMENT TOOLS	12
FLASH MEMORY CHIP PROGRAMMING PROCEDURE	12
PAL PROGRAMMING PROCEDURE	12
MIUW-SPECIFIC DETAILS	13
DRAWINGS	15
APPENDICES	
A: PAL FIRMWARE SOURCE CODE	A-1
B: MICROCONTROLLER SOURCE CODE	B-1

Figures

1. VIS Controller Board schematic	3
2. VIS Controller Board wiring diagram	14
3. Board layout silkscreen. Scale: 2:1	16
4. Top layer of copper	17
5. Bottom layer of copper	18
6. Vcc layer (power plane). Scale: 2:1	19
7. GND plane layer. Scale: 2:1	20

Tables

1. VIS Controller Board parts list.....	4
2. PAL outputs.....	9

BACKGROUND

The U.S. Navy's Mobile Inshore Undersea Warfare (MIUW) system is an upgrade program for the Naval Reserves that provides enhanced surveillance and communication capabilities for port security, harbor defense, and the coastal warfare mission. This system requires a video camera and originally used a black and white camera. SPAWAR Systems Center, San Diego (SSC San Diego) upgraded to a color camera for better target detection and identification. For ease of integration, the color system needed to be totally compatible with the original system without requiring changes to hardware or software. After determining that no commercial off-the-shelf products were available that were

compatible with the old software, modification of a product was considered. The video camera manufacturer offered to modify the camera for a fee. SSC San Diego determined that it would be more cost effective for the Navy to design and install a basic interface board in the off-the-shelf camera assembly.

This report describes the Video Imaging System (VIS) interface board that was designed by SSC San Diego and deployed in MIUW cameras.

BOARD REQUIREMENTS

The VIS interface board is inserted between the camera and the MIUW computer. The MIUW computer uses an RS-422 interface to issue commands in a format peculiar to the original black and white camera. These commands control camera power, zoom, focus, and iris. However, current camera interface command formats bear little resemblance to those used in the original MIUW camera assembly. Therefore, the board must translate the commands. The board must also interpret lens and power commands and include switches for these functions.

The camera requires lens feedback. This function requires a hardware solution apart from the camera, since no cameras provide a zoom/focus motor and preset interfaces. The hardware solution must read zoom and focus motor positions and send the information back to the interface.

The MIUW application must remotely alter iris settings through the serial interface. Some cameras offer a video output that automatically controls the iris. For remote control, two options are available. Either intercept the signal and modify it, or command the camera to change the signal. The first option requires hardware.

VIDEO IMAGING SYSTEM (VIS) INTERFACE BOARD

The VIS interface board provides wide flexibility in the camera interface. It employs a Siemens version of the popular 8051 type of microcontroller that has two Universal Asynchronous Receiver-Transmitter (UART) serial interfaces so it can communicate simultaneously with the user and a digital camera. This allows translation of commands from one command set to another so the interface can be made compatible with any digital camera with a serial interface and any user command set.

This board includes relays for controlling the zoom and focus motors of the lens and for turning the camera power on and off.

Firmware is stored in non-volatile flash, electrically erasable programmable read only memory (EEPROM) and modified remotely. This is accomplished by logic that switches the microcontroller to run out of volatile random access memory (RAM) during reprogramming.

The microcontroller contains an analog-to-digital converter for zoom feedback and focus settings.

If fully populated, this board uses a video frequency, multiplying digital-to-analog converter (MDAC) to modulate the automatic iris feedback for controlling the relative iris position. This enables the user to control the feedback loop to increase or decrease the light gathered by the lens without turning off the automatic iris function.

INTERFACE REQUIREMENTS

HARDWARE

The board must interface with the MIUW Interface Processor (IP), the camera, and the lens. Connector designations in this section refer to the board schematic. Figure 1 shows a schematic of the VIS Controller Board. Table 1 is the parts list.

Power is supplied to the board through a 2-pin connector (J1) and is nominally 12 volts DC. Anything from 7 to 15 volts will work, although 12 volts should be maintained for consistent zoom timing.

The camera and interface protocol (IP) serial interfaces are RS-422 standard interfaces using 4-pin connectors (J5 and J9).

The lens zoom and focus require a positive voltage to zoom or focus in one direction and a negative voltage to zoom or focus in the other direction. The 12 volts DC is routed to them through combinations of six relays and 3-pin connector J3 to enable a single supply to apply positive or negative voltage to either motor. Note that it is important to isolate the motors from ground for this configuration.

Preset potentiometers provide zoom and focus position feedback. Ground and 5 volts are provided, and wiper voltages are read through a 4-pin connector, J8.

Camera power uses a separate 12-volt supply that this board must control. The power comes into this board through a 2-pin connector, J2, and passes out to the camera through a 2-pin connector, J4. A relay on the board switches power on or off.

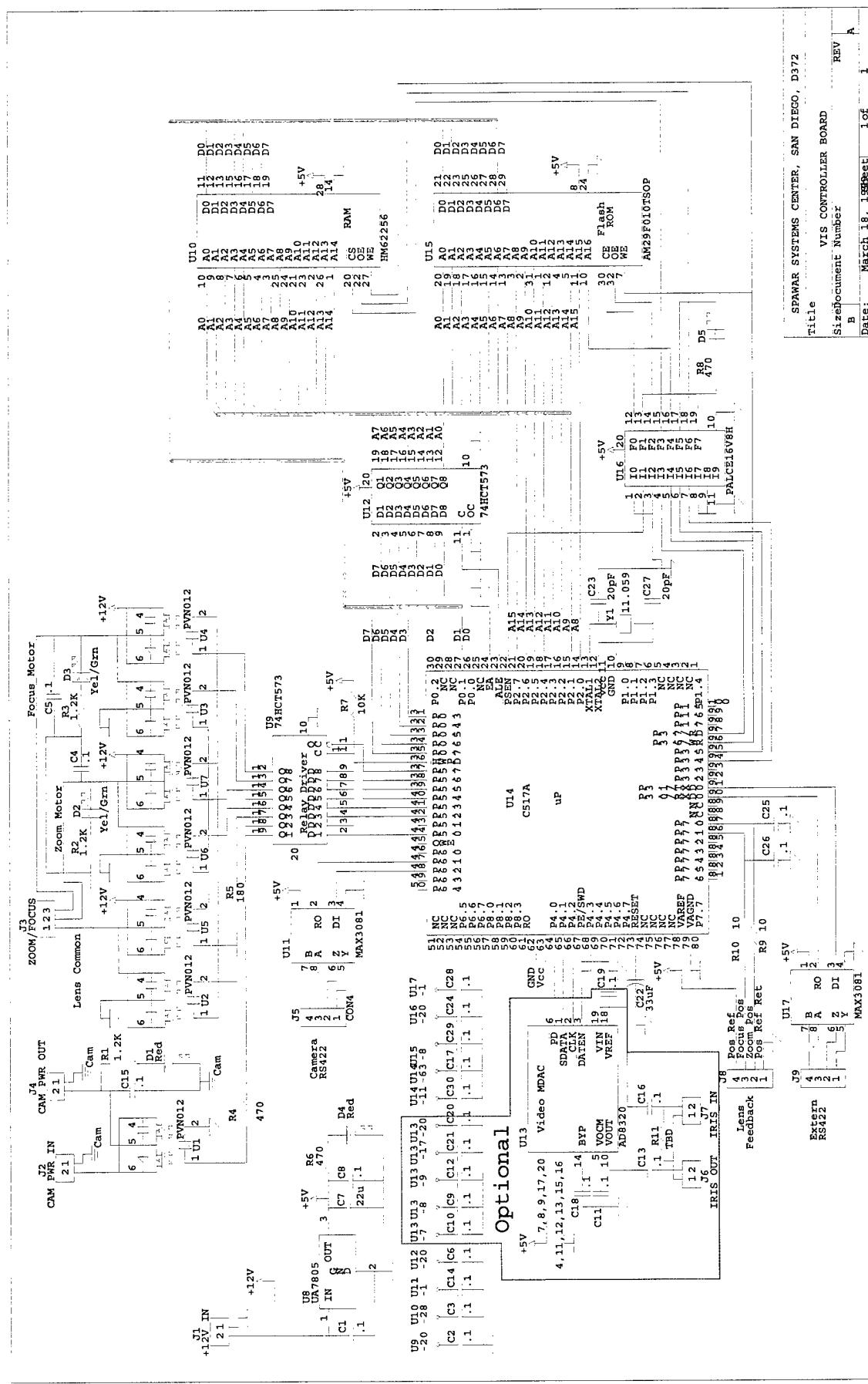


Figure 1. VLSI Controller Board schematic.

Table 1. VIS Controller parts list.

Reference Designator	Description	Manufacturer	Part Number	Qty	Source
U1, U2, U3, U4, U5, U6, U7	SS Relay	IR	PVN012S	7	Arrow/Zeus (949) 454-4275
U8	5V Regulator	TI	UA7805QKC	1	Arrow/Zeus
U9, U12	Data latch	TI	74HCT573DW	2	Arrow/Zeus
U10	SRAM	Hitachi	HM62256BLFPI-7	1	Insight (619) 677-3100
U11, U17	RS232 I/F	Maxim	MAX3081ESA or MAX488EESA	2	Maxim (800) 998-8800
U14	Microcontroller	Siemens	SAF-C517A-LM	1	Insight
U15	Flash EPROM	AMD	*AM29F010-70EI	1	Arrow/Zeus
U16	PAL	Vanits	*PALCE16V8H10PI/4	1	Arrow/Zeus
Y1	Xtal, 11.059 MHz	Epson	Model MA-505 SE2506CT-ND	1	DigiKey (800) 344-4539
D1, D4	LED, Red, 3mm		351-3232	2	Mouser (800) 346-6873
D2, D3, D5	LED, Grn/Yel	King Bright	604-L937GYW	3	Mouser
C1, C2, C3, C4, C5, C6, C8, C14, C15, C17, C24, C25, C26, C28, C29, C30	Capacitor, 0.1 μ F, ceramic				
C7, C22	Capacitor, 33uF, solid tantalum	Thomson	581-33M10V	2	Mouser
C9, C10, C11, C12, C13, C16, C18, C19, C20, C21	Unused capacitor designations			0	
C23, C27	Capacitor, 22pF, ceramic	Mallory	539-CK05220K	2	Mouser
R1, R2, R3	Resistor, 1.2K, 1/8W, 5%	Yageo	1.2KE BK-ND	3	DigiKey
R4, R6, R8	Resistor, 470 ohm, 1/8W, 5%	Yageo	470E BK-ND	3	DigiKey

Table 1. VIS Controller parts list. (continued)

Reference Designator	Description	Manufacturer	Part Number	Qty	Source
R5	Resistor, 180 ohm, 1/8W, 5%	Yageo	180E BK-ND	1	DigiKey
R7	Resistor, 10K, 1/8W, 5%	Yageo	10KE BK-ND	1	DigiKey
R9, R10	Resistor, 10 ohm, 1/8W, 5%	Yageo	10E BK-ND	2	DigiKey
J1, J2, J4	Connector, 2-pin	JST	B 2B-PH-K	3	JST (800) 947-1110
J3	Connector, 3-pin	JST	B 3B-PH-K	1	JST
J5, J8, J9	Connector, 4-pin	JST	B 4B-PH-K	3	JST
J6, J7	Unused connector designations				

* IMPORTANT: The Flash EPROM and PAL (U15 and U16) must be programmed before installation!

SOFTWARE

The camera software interface is described in the Cohu "Installation and Operation Instructions" document for the "Model 3500 DSP Interline Transfer Color Camera." The interface to the MIUW computer, also referred to as the Interface Processor or IP, VIS Interface Design Document (IDD) as follows:

VIS IDD November 22, 1998

1.0 Physical Connections.

4 wire (plus ground) RS422, full duplex

9600 baud, 8 bits data, one stop bit, no parity

2.0 Message Formatting.

2.1 Computer to VIS

Messages sent from the IP computer to the VIS camera are always formatted as follows:

STX (0xF8) Start of Transmission

Camera # Number of Selected Camera (0x01 always)

Camera command and data

Checksum byte

The checksum is calculated in a strange way. It is computed by performing an exclusive 'OR' of all the bytes in the message, following the STX character, and storing only the lower

nybble (lower 4 bits) of that value, with the most significant bit set. The following C code shows the method the VIS camera calculates the checksum byte:

```
Calc_checksum(unsigned char *ptr, count)
{
    int ii, xx;

    xx = 0;
    for(ii = 0; ii < count + 1; ii++) {
        xx ^= ptr[ii];
    }
    *ptr[ii] = 0x80 | (xx & 0x0F);
}
```

When a valid message is received at either the VIS or MSP, the receiver responds with an 'ACK' character (0x06). If a message is detected with a bad checksum or other error, the receiver responds with a NACK (0x15). This response precedes any response to the received message (i.e., the ACK is sent before the response to a status request, for example).

2.2 Camera Commands used by MSP Software

Toggle commands are two character commands that toggle the selected feature in the camera.

'LP' - Toggle Power
'LM' - Toggle Autoliris

Iris Open/Close/Stop commands are sent to cause the camera to open or close the iris. These are sent only if not in auto iris mode:

'IO' - Iris Open
'IS' - Iris Stop
'IC' - Iris Close

Status Inquiry commands are sent periodically. These are used to keep the MSP software up to date on the lens positions and other states. The 'On Line' indicates the fact that status messages are being received by the MSP indicator on the MSP VIS control panel.

'L?' - Request a 'Latch Status'
'V?' - Request a Lens Position Status

To set the lens zoom position and focus position, the following command is sent:

'v' - Lens Position Request. This is followed by six bytes. The first three bytes represent the zoom position, the last three bytes is the focus position. These position values

represent a 12-bit range from 0 to 4095. 4095 represents max zoom (smallest field of view) and focus to nearest range. The position is encoded four bits at a time, most significant bits first, each 'OR'd with the value 0 x 30. The following code illustrates creating a lens position message:

```
*ptr++ = 'v';
*ptr++ = ((zoompos & 0x0F00) >> 8) | 0x30;
*ptr++ = ((zoompos & 0x00F0) >> 4) | 0x30;
*ptr++ = (zoompos & 0x000F) | 0x30;
*ptr++ = ((focuspos & 0x0F00) >> 8) | 0x30;
*ptr++ = ((focuspos & 0x00F0) >> 4) | 0x30;
*ptr++ = (focuspos & 0x000F) | 0x30;
```

2.2 VIS to Computer

VIS messages are formatted exactly as described for MSP messages to the VIS. The MSP software decodes the following VIS messages.

'L' - Latch Status. Three bytes of status follow, with only the first byte being decoded. The lower nybble is decoded as follows:

Bit 0 – Iris Status. If 1, iris is manual, if 0, iris is automatic.

Bit 1 – Camera Power. If 1, camera is powered. If 0, camera is off.

The other bits and bytes are ignored.

'V' - Lens Status. Six bytes follow, formatted exactly as the lens position command.

HARDWARE DESIGN

The first concern in designing the board was keeping the design simple to minimize components and keep the board small enough to fit in a 4-inch camera housing, if desired. The Siemens microcontroller contains two UART serial interfaces and an analog-to-digital converter with eight multiplexed inputs. Having these features integrated into the microcontroller chip simplifies the board design and reduces the chip count.

Firmware for the microcontroller is contained on a Flash EEPROM chip (AMD AM29F010). During use, the firmware is copied to a RAM chip (Hitachi HM62256) and execution takes place out of the RAM. The 8051 microcontroller uses a "Harvard Architecture," which means that the program and data memory share the same address space. Therefore, the board uses a programmable logic array (PAL) to control which chip is mapped as program and which is mapped as data memory. This allows the Flash ROM to be reprogrammed by the microcontroller under remote control. The advantage of this arrangement is that the firmware can be easily modified with no disassembly of hardware. The disadvantage is that it requires a RAM chip and a PAL and added software complexity.

A 74HCT573 octal latch provides address latching for memory access. This is required since the microcontroller multiplexes the data and lower address signals onto eight pins.

Another 74HCT573 octal latch is used as a buffer between the microcontroller digital output pins and the relay inputs. The relays are solid state International Rectifier PVN012 optocoupled AC/DC relays. Two relays are used for each of the three pins: Zoom, Focus, and Common. One of the two relays can connect the pin to ground, and the other to +12 volts. This configuration allows a single +12-volt supply to provide positive or negative voltage to drive the zoom or focus motors. It is important to note that this will only work if the motors are isolated from ground.

A seventh PVN012 relay is dedicated to controlling camera power.

Two Maxim MAX3081 interface chips convert serial interface (UART) signals at the microcontroller to RS-422 signals at the camera and IP.

A UA7805, standard, 3-pin, 5-volt regulator provides logic power. An 11.059-MHz crystal for the system clock ensures that the appropriate serial interface frequency can be derived. A 33-microfarad capacitor at the microcontroller reset pin provides a solid power-up reset.

An optional feature is enabled when an Analog Devices AD8320 video multiplying digital-to-analog converter (MDAC) is installed. With this device, the amplitude-modulated video signal from the camera controlling the lens iris can be increased or decreased to open or close the iris. This is considered optional, because after the first board was built and tested for MIUW, it was determined that Cohu had changed their camera design to use a pulse-width-modulated video signal instead of an amplitude-modulated signal. This feature was not installed on subsequent boards. Instead, direct camera commands are issued to alter picture brightness. This produces the same results, but uses the camera digital signal processor.

Also optional on the board are five light emitting diodes (LEDs). These are used in testing and debugging the board. As configured in the schematic, the LEDs indicate the following:

D1 – Red when camera power is turned on

D2 – Yellow when zooming in, green when zooming out, off otherwise

D3 – Yellow when focusing near, green when focusing far, off otherwise

D4 – Red anytime power is applied to board

D5 – Yellow when executing program out of RAM, green when executing out of high Flash memory, off otherwise

Note: The green and yellow of D2 and D3 may be reversed, depending on the lens motor wiring.

FLASH REPROGRAMMING

The software controls flash memory reprogramming. This section describes the signals and logic involved in the memory control on the board, for use by someone making software or hardware modifications to the board. For a description of the reprogramming procedure used for altering the first software version, see the section entitled **FLASH REPROGRAMMING PROCEDURE**.

When power is first applied, the RAM will be empty, so the microcontroller must execute the firmware in the flash memory. To reprogram the flash memory, the microcontroller must be executing firmware out of RAM and accessing the flash as data memory. Therefore, the hardware must de-select flash and select RAM by switching the chip control signals in the middle of an instruction execution, without causing an execution glitch.

To do this, the Program Store Enable (PSEN) signal, Read (RD) signal, and two controllable port pins generate Chip Select (CS)/Chip Enable (CE) signals, Output Enable (OE) signals and a high address (A16) signal. One of the controllable port pins (P3.5) is designated as the flash programming selector (PGM). The other (P3.4) is designated the A16 address bit, inverted for proper reset initialization.

The signal logic is described in the following subsections.

PAL INPUTS

The PGM pin (P3.5) is made active (low) to command the system to execute out of RAM.

The A16 pin (P3.4) is made active (low) to select high flash memory.

PAL inputs also include the microcontroller RD and PSEN signals.

PAL OUTPUTS

The OE outputs carry the RD and PSEN signals. If in flash programming mode (PGM is active), then the flash OE = RD and the RAM OE = PSEN. Otherwise, the opposite applies.

The CE/CS outputs carry the PSEN and its complement. If in flash programming mode, the RAM CS = PSEN and the flash CE = PSEN inverted. Otherwise the opposite applies.

The A16 output is the complement of the A16 input. Table 2 also lists PAL output information.

Table 2. PAL outputs.

PAL Outputs	Normal Mode (PGM Not Active)	Flash Programming Mode (PGM Active)
Flash OE	PSEN	RD
RAM OE	RD	PSEN
Flash CD	PSEN	PSEN complement
RAM CS	PSEN complement	PSEN
A16	A16_ input complement	A16_ input complement

For added protection from disaster during programming, only the higher part of the flash chip is reprogrammed. The lower part is reserved for the basic reprogramming software. In case of error in programming, the software will still be able to run out of lower memory to accept instructions to reprogram the higher memory.

For another approach to in-system flash programming, see the Siemens Semiconductors application note AP0821 "C5XX/80C5XX In-System FLASH Programming" on their CD-ROM entitled "8-bit and 16-bit Microcontrollers CD-ROM Edition 2.0, *Siemens Semiconductors*, November 1997.

FLASH REPROGRAMMING PROCEDURE

To reprogram flash memory, the board must be powered up and connected to a PC via a serial port. Since the board uses an RS-422 interface and most PCs only have RS-232 serial ports, a converter such as the Telebyte Model 231 must be inserted.

The commands for reprogramming the flash memory were designed to use the "WinBurner Flash Memory Burner" software version 1.0.18, produced by Automated Enterprises, LLC, Scott M. Olmsted, 760-942-1953. This program runs in Windows 95 and sends a file using a logical sequence of handshaking communication through the serial port. Selections required for this program are:

Com1, 9600 baud, then click on "Open"

Check the boxes for: Insert Start Field, Insert Length Byte, Insert Stop Field, Insert 1-byte Checksum, Length Includes Checksum

Start: F8

Command: 55 Erase

66 Record

77 End

Stop: 03

Select Erase Type: App 1

Ack: 06

File Name: c:\vis\vis\vis.hex {this reflects the location of the compiled code.}

All other boxes and fields are left blank.

After setting up the software as described, click on "Download" to reprogram the flash memory. These settings are likely to be different if later versions of the WinBurner software are used, since this is an early (alpha test?) version and the byte counts seem to be off by one. But, this version has been successfully used with these settings.

The sequence of bytes sent is:

Start (F8h)

Length (1 byte)

Command (55h or 66h or 77h)

Type (0 for boot sector or low flash addresses, 1 for application sector or high flash)

Stop (03h)

Checksum (1 byte)

The checksum is a simple modulo 100h sum of all preceding bytes.

SOFTWARE DESIGN

Firmware is required in two locations on this board. The microcontroller firmware is located in the Flash ROM. PAL firmware is located on the PAL chip.

The PAL firmware is quite simple and implements the algorithms defined in the flash reprogramming section above. Appendix A provides the PALASM source code.

The microcontroller firmware was derived from original code developed by Gary Perkins of MicroWare Technology, Inc., San Diego, California. The source code was written in C and assembly language.

The assembly language code contains subroutines for copying code from flash to RAM and subroutines for switching back and forth between executing out of flash and executing out of RAM.

The main C routines are in the “Ric.c” file. These control the major functions as well as various minor functions.

The “Flash.c” file contains the routines for reprogramming the flash memory, putting out signals and commands to erase and reprogram.

The “Hostmsg.c” file contains routines which handle commands received from the IP serial port.

The “Ricisr.c” file contains the interrupt subroutines.

Files required for code compilation are:

C source code:

Ric.c

Flash.c

Hostmsg.c

Ricisr.c

Ric.h

Proto.h

Ricglob.h

Assembly language source code:

Ricasm.a51

Visboot.m51

Utility file:

M.bat

Appendix B provides printouts of the code.

SOFTWARE DEVELOPMENT TOOLS

The 8051 development tools chosen were Keil Software CA51 V5.2 Compiler Kit, comprising a C compiler and an 8051 assembler.

The PAL development tool used was a PALASM4 Ver 1.5 compiler, available as freeware from Lattice Semiconductor/Vantis at <http://www.latticesemi.com>.

FLASH MEMORY CHIP PROGRAMMING PROCEDURE

Although any compatible programmer can be used, the following is a procedure for using the Advin Systems, Inc. Pilot-U40. Filenames and directories may vary, but this procedure is based on the original computer configuration. If the software is modified, the checksum will be different.

1. Connect the Pilot-U40 to computer parallel port and apply power.
2. Execute program “spee.exe” in the Advin directory.
3. Set device to AMD 29F010.
4. Set filename to c:\vis\vis\vis.hex
5. Load file into buffer and program chip using menu commands.
6. Verify that checksum is 7C46.

PAL PROGRAMMING PROCEDURE

Although any compatible programmer can be used, the following is a procedure for using the Advin Systems, Inc. Pilot-U40. Filenames and directories may vary, but this procedure is based on the original computer configuration.

1. Connect the Pilot-U40 to computer parallel port and apply power.
2. Execute program “spgal.exe” in the Advin directory.
3. Select AMD/MMI PALCE16V8 as the device type.
4. Set filename to c:\advin\palasm\visfil~1\vis.jed
5. Load file into buffer and program chip using menu commands.

6. Verify that checksum is 3475.

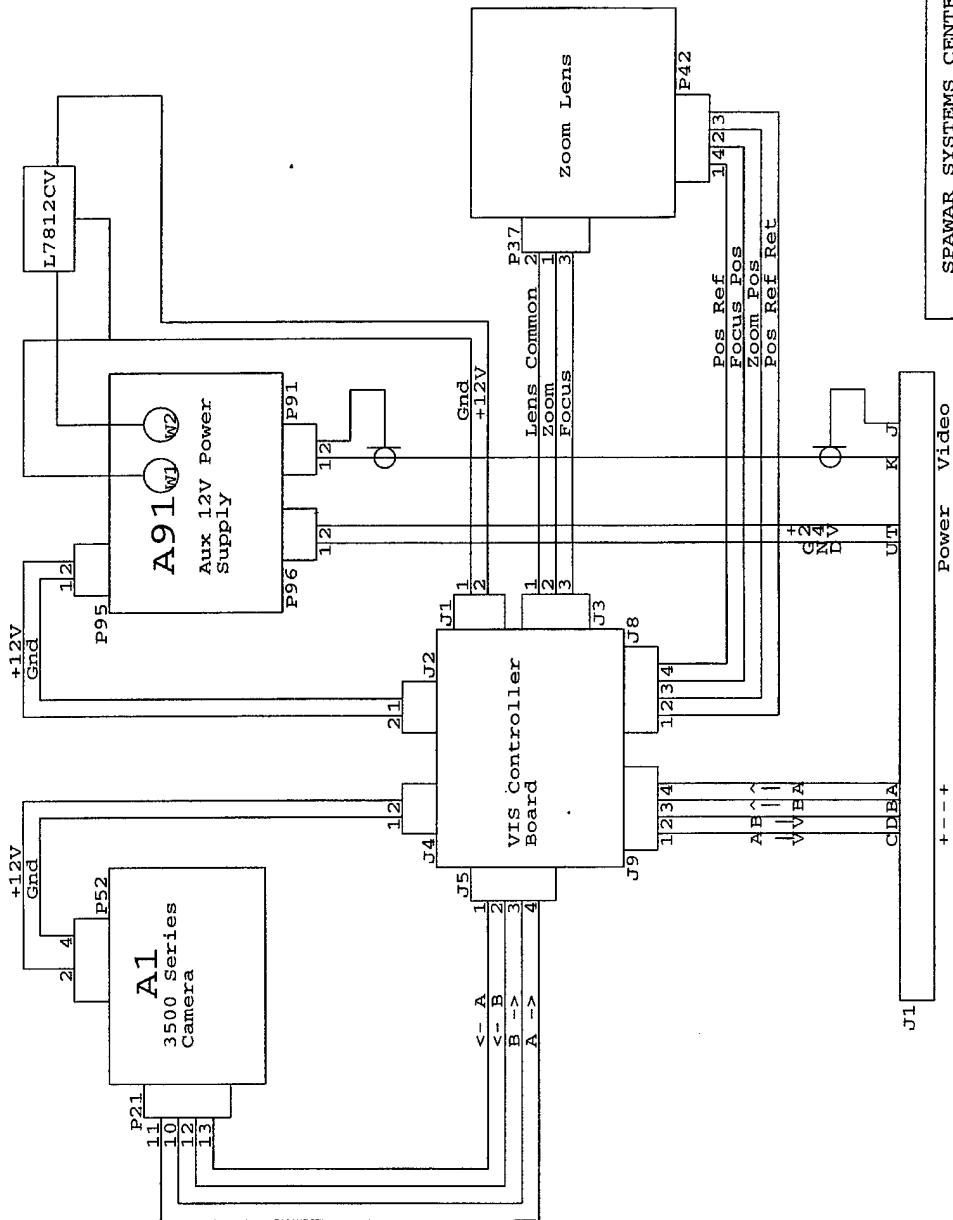
MIUW-SPECIFIC DETAILS

For the MIUW application, the Cohu Model 3522 color video DSP camera with engineering revision ER4565HN (Cohu's internal number) is used. This uses the Cohu Model 3522-1000 DSP interline transfer CCD color camera with a ½-inch sensor, NTSC compatibility, and a 12-volt DC supply. The lens is the Cohu "Model P10K," 10X, F1.8, 16 to 160 mm, with zoom and focus presets. Also included is a 2X range extender.

Figure 2 shows the wiring of the board to the camera.

For details of the first procurement, see contract N66001-99-C-0034.

Notes:
 A81, Zoom Lens Driver Board required but not shown.
 This diagram only shows those connections related to the VIS Controller Board.
 All other pins on J1 are unused. The heater must be connected to pins T and U.



SPAWAR SYSTEMS CENTER, SAN DIEGO, D372	
Title	VIS Controller Board Wiring Diagram
Size	Document Number
A	REV
Date:	May 27, 1999 Sheet 1 of 1

Figure 2. VIS Controller Board wiring diagram.

DRAWINGS

For reference, figures 3 through 7 show the printed wiring board layout drawings. There are two jumpers required on the board and one area of difficulty the manufacturer had in populating the boards using automatic techniques.

The printed wiring board requires the following two jumpers to match the schematic:

1. U14 pin 36 (currently open) to U9 pin 11
2. U14 pin 67 (currently open) to GND (the nearest C20 pad)

When populating the boards, carefully inspect the solder joints at the points that have feed-through holes reducing the pad size. Depending on the tolerances used in the etching process, the pad size may be very small, and it may be necessary to add solder after the automated soldering operation.

The pads affected are:

- U1 pin 2
- U2 pin 5
- U3 pin 5
- U6 pin 5
- U10 pin 14
- U12 pins 1 and 10
- U17 pin 4

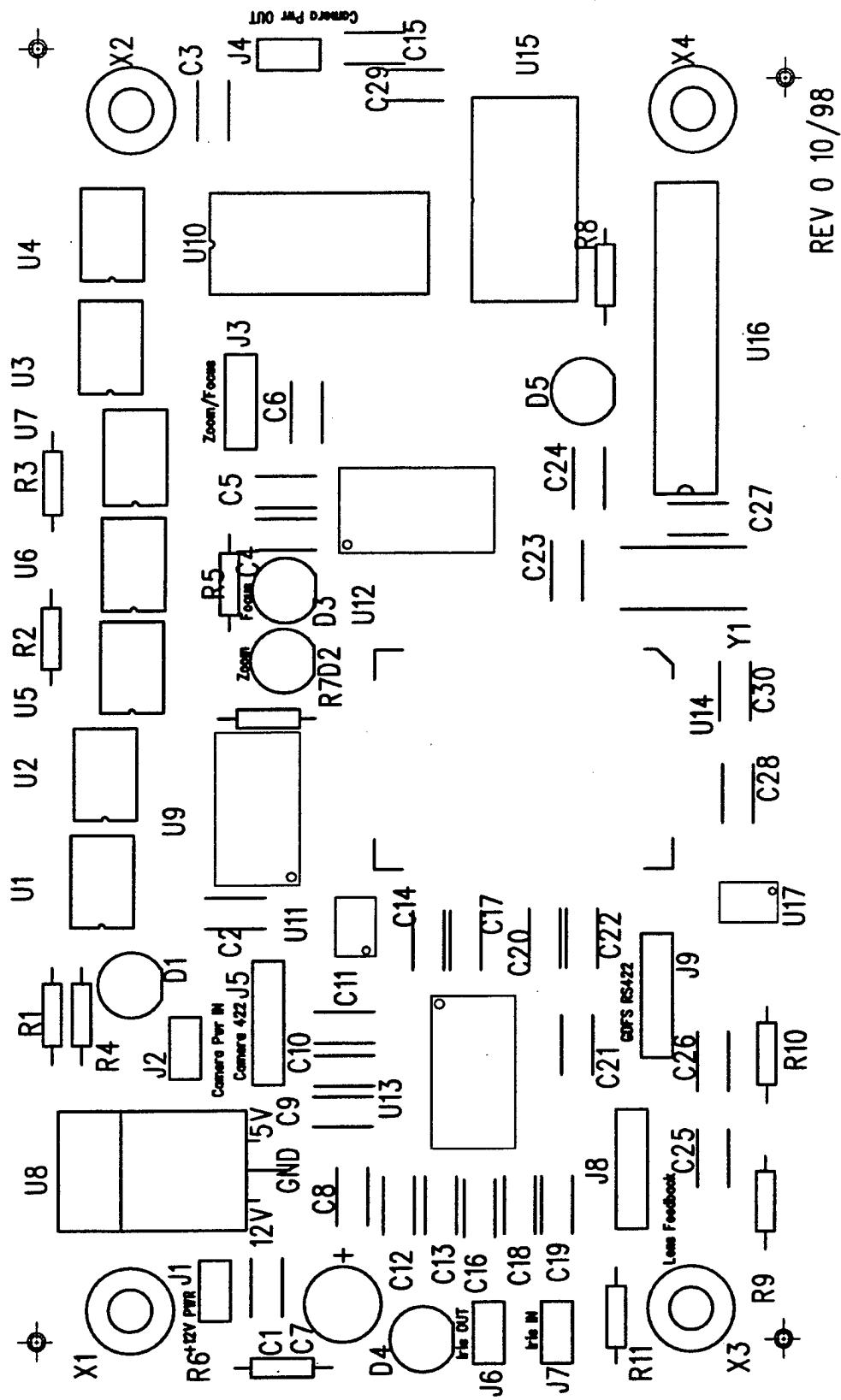


Figure 3. Board layout silkscreen. Scale: 2:1

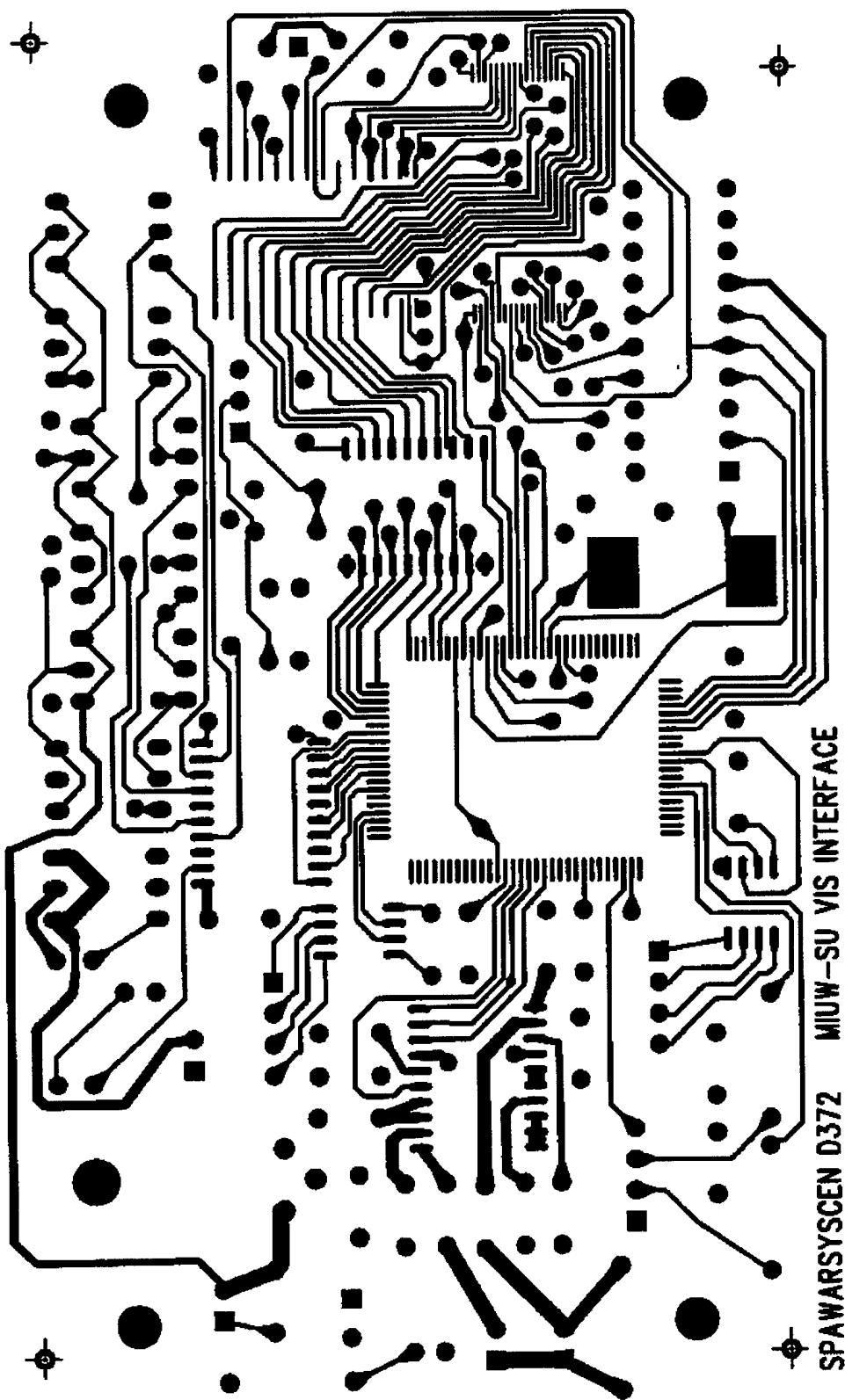


Figure 4. Top layer of copper.

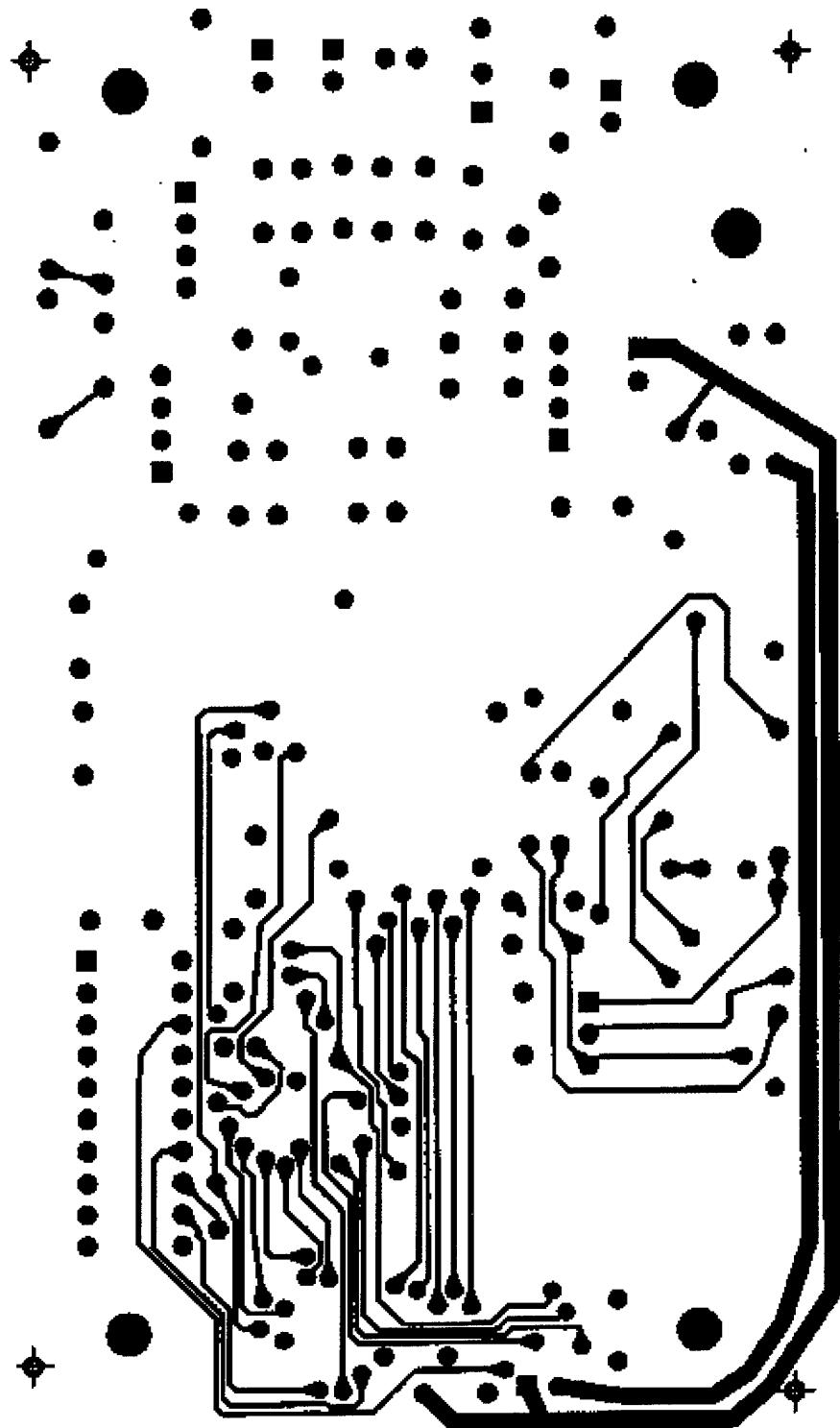


Figure 5. Bottom layer of copper.

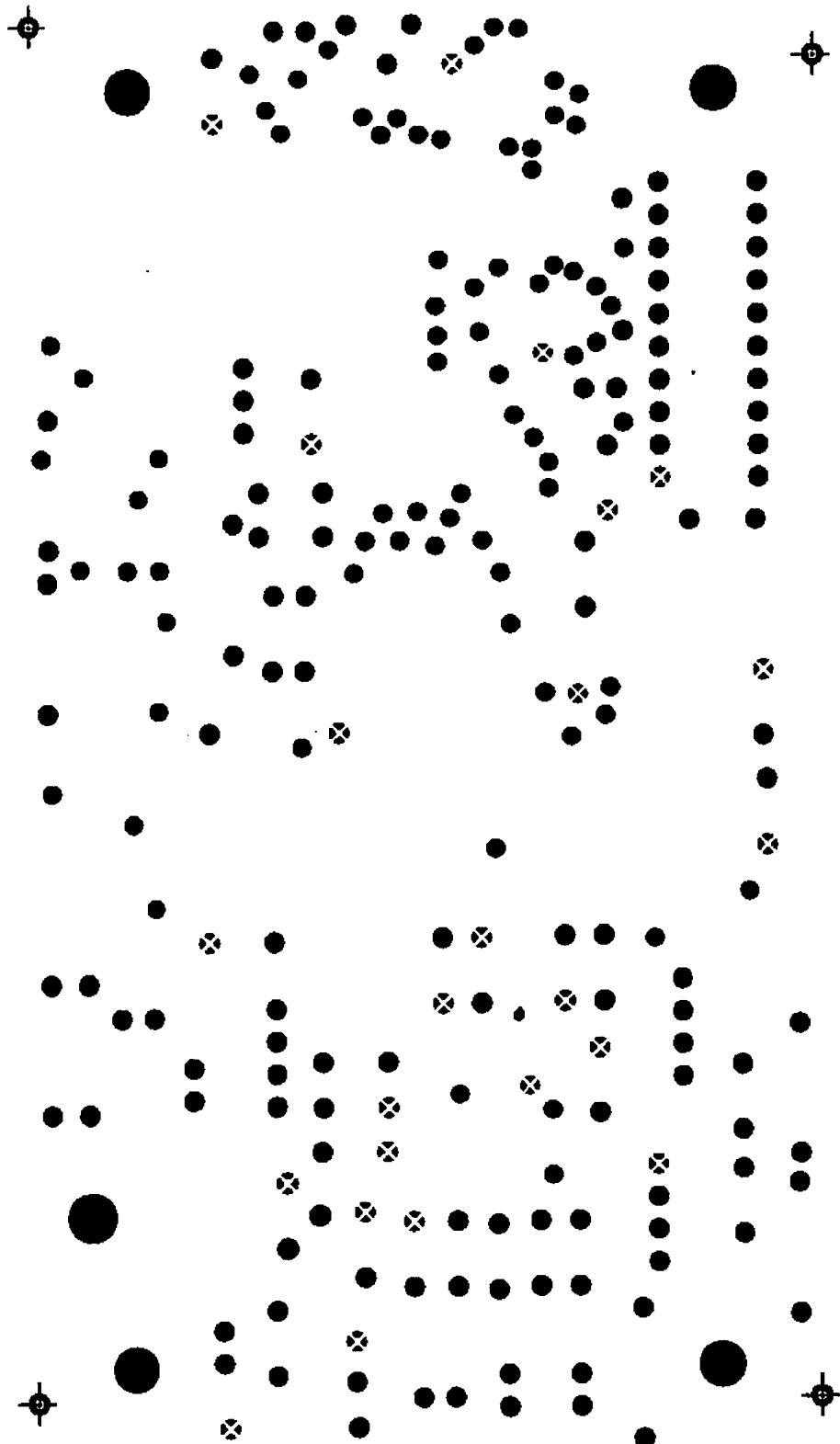


Figure 6. Vcc layer (power plane). Scale: 2:1.

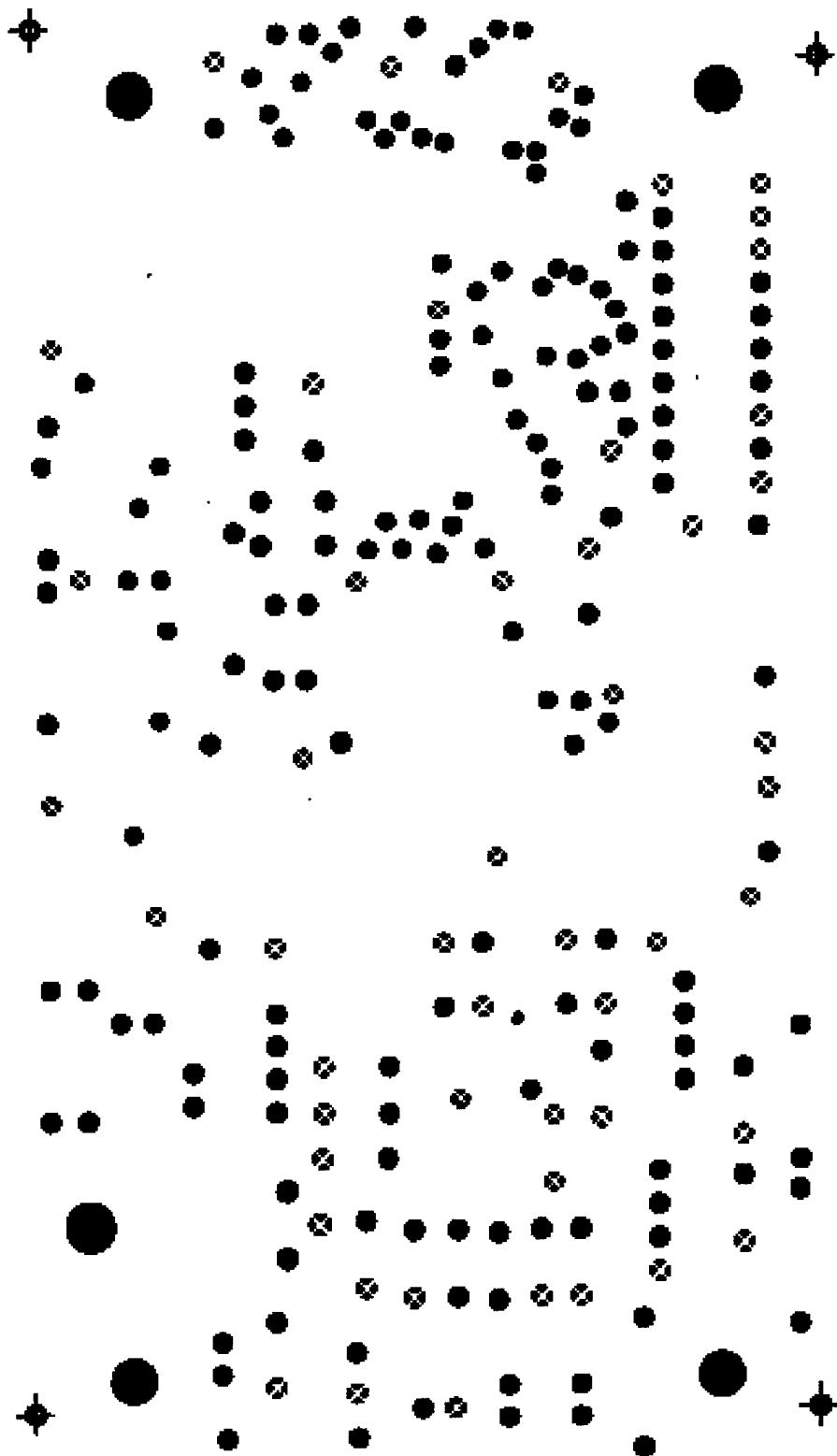


Figure 7. GND plane layer. Scale: 2:1.

APPENDIX A

PAL FIRMWARE SOURCE CODE

```

; ****
; This is for the MIUW VIS Controller PAL.
;
; ****
; Allowable Target Device Types: PALCE16V8H-10
; ****

TITLE          Vis Controller PAL
REVISION       -
PATTERN        ?
AUTHOR         Jeff Coleman
COMPANY        SSC-SD
DATE          1/8/1999

CHIP          VIS_U16      PALCE16V8H-10

; *  Inputs  **
PIN 2  PSEN_      ; Program Store Enable NOT      *
PIN 4  RD_         ; Read NOT                      *
PIN 5  WR_         ; Write NOT                     *
PIN 6  PGM_        ; Program Enable NOT (Port 3.5)  *
PIN 7  A16_        ; A16 NOT (Port 3.4)          *

; *  Outputs  **
PIN 12 FLASH_CE_  ;                                *
PIN 13 LEDa       ;                                *
PIN 14 FLASH_OE_  ;                                *
PIN 15 RAM_CS_    ;                                *
PIN 16 A16         ;                                *
PIN 17 LEDb       ;                                *
PIN 18 RAM_OE_    ;                                *

EQUATIONS

;          Normal mode      Programming mode *
FLASH_OE_   =      (PGM_ * PSEN_) + (/PGM_ * RD_)
RAM_OE_     =      (PGM_ * RD_) + (/PGM_ * PSEN_)
FLASH_CE_   =      (PGM_ * PSEN_) + (/PGM_ * /PSEN_)
RAM_CS_     =      (PGM_ * /PSEN_) + (/PGM_ * PSEN_)

A16 = /A16_      ; The A16 address bit is P3.4 inverted
LEDa = /A16_      ; LED will be off when running in low FLASH, or on in
LEDb = /PGM_      ; green or yellow, indicating whether running in high
                  ; FLASH or low RAM.

; EXPLANATION OF LOGIC
; "Program Store Enable" (PSEN_) is active when reading external
; program memory. It is routed to Flash when Program mode is not
; selected (PGM_ is high) and to RAM when Program mode is
; selected (PGM_ is low).
;
; RD_ is routed to RAM_OE_ when Program mode is selected
; (PGM_ is low) and FLASH_OE_ when Program mode is not selected.
;
; PSEN_ is inverted and routed to RAM_CS_ when Program mode is
; selected (PGM_ is low) and FLASH_CE_ when Program mode is
; not selected.

SIMULATION

```

```
TRACE_ON PSEN RD PGM A16 LEDa LEDb A16 FLASH_OE FLASH_CE RAM_OE RAM_CS
SETF PSEN RD WR PGM A16;
SETF PSEN /RD /WR PGM A16;
SETF /PSEN /RD WR PGM A16;
SETF /PSEN /RD /WR /PGM /A16;
SETF PSEN /RD WR /PGM /A16;
SETF /PSEN /RD /WR /PGM /A16;
TRACE_OFF
```

APPENDIX B

MICROCONTROLLER SOURCE CODE

C source code:

```

***** C source code *****

/*
 * Name: RIC.C
 *
 * Functions:
 *     main();
 *     init_globals();
 *     init_sys()
 *     echo_setup();
 *     flag_error();
 *
 * Description:
 *
 *     This module contains main() which is the first procedure
 *     called after reset. (This handles the FLASH checking/downloading)
 *
 * History: Adapted for VIS (12/2/98)
 */

***** C source code *****

#include "ric.h"           /* Constants, hardware definitions, etc. */
#include "proto.h"          /* Function prototypes for all modules */
#include "ricglob.h"         /* Global variable (externs) definitions */

// This buffer is declared public in hostmsg.c (optimizing the use of internal
// data RAM) and then precedes the bit area so there is no gap in data space
extern tx_msg_header HOST_Txmsg_header;      // Header building area for Host msgs

***** GLOBAL VARIABLES *****

bit sending_ACK;           // True when ACK is being sent to host
bit HOST_ACK_timeout;      // Indicates that the Host ACK timer expired
bit CAMERA_ACK_timeout;    // Indicates that the Camera ACK timer expired
bit HOST_Rx_timeout;       // Timeout flag for Host Rx data stopping

bit sending_to_camera;     // Host is sending to camera (started in APP)
bit wait_for_T0_timeout;   // 1 msec tick generator
                           // Used by flash programming functions
                           // Flash programming block
                           // Image of data written to Port 3
                           // Timeout counter for Host Rx (bad rx msg)
                           // Timeout counter for Camera Rx (bad rx msg)
                           // 1 msec tick timer/counter
int digital_zoom_in_timer; // 1 msec digital zoom timer/counter
int digital_zoom_out_timer; // 1 msec digital zoom timer/counter
                           // Counter keeping track of reply retries
                           // State of the Rx data stream from Host
                           // State of the Tx data stream to Host
                           // Keeps track of the byte count
                           // Keeps track of the byte count
                           // Keeps a running tally of checksum
                           // Keeps a running tally of checksum

unsigned char RF_Txmsg_retries; // State of the Rx data stream from CAMERA
unsigned char HOST_Rxmsg_status; // State of the Tx data stream to CAMERA
unsigned char HOST_Txmsg_status; // Keeps track of the byte count
unsigned char HOST_Rxmsg_byte_cnt; // Keeps track of the byte count
unsigned char HOST_Txmsg_byte_cnt; // Keeps a running tally of checksum
unsigned char HOST_Rxmsg_checksum; // Keeps a running tally of checksum
unsigned char HOST_Txmsg_checksum; // Keeps a running tally of checksum

unsigned char CAMERA_Rxmsg_status; // State of the Rx data stream from CAMERA
unsigned char CAMERA_Txmsg_status; // State of the Tx data stream to CAMERA
unsigned char CAMERA_Rxmsg_byte_cnt; // Keeps track of the byte count
unsigned char CAMERA_Txmsg_byte_cnt; // Keeps track of the byte count
unsigned char CAMERA_Rxmsg_checksum; // Keeps a running tally of checksum
unsigned char CAMERA_Txmsg_checksum; // Keeps a running tally of checksum

```

```

int last_dnload_msg_state;           // State of RF dnload msgs (ignore retries)
int HOST_ACK_timer;                // Host ACK timer (keeps track of reply timing)
int CAMERA_ACK_timer;              // Camera ACK timer (keeps track of reply timing)
unsigned char idata *HOST_Rxmsg_byte_ptr; // Points to current position in Rxmsg_buf
unsigned char idata *HOST_Txmsg_byte_ptr; // Points to current position in Txmsg_buf
GDFS_Rxmsg_buf idata HOST_Rxmsg_buf; // Host Rxmsg buffer (data from host)
GDFS_Txmsg_buf idata HOST_Txmsg_buf; // Host Txmsg buffer (data to host)

unsigned char idata *CAMERA_Rxmsg_byte_ptr; // Points to current position in Rxmsg_buf
unsigned char idata *CAMERA_Txmsg_byte_ptr; // Points to current position in Txmsg_buf
Cam_Rxmsg_buf idata CAMERA_Rxmsg_buf; // CAMERA Rxmsg buffer (data from CAMERA)
Cam_Txmsg_buf idata CAMERA_Txmsg_buf; // CAMERA Txmsg buffer (data to CAMERA)

unsigned char zoom_status;          // Latest measured value of zoom
unsigned char focus_status;         // Latest measured value of focus
int zoom_cmd;                      // Decoded 12 bit GDFS zoom command
int focus_cmd;                     // Decoded 12 bit GDFS focus command
unsigned char latch_status;         // Bit 0 - Iris status. 1=manual, 0=automatic
                                    // Bit 1 - Camera Power. 1=on, 0=off
unsigned char iris_setting;         // Iris gain setting
unsigned char zoom_AD;              // Value desired for zoom A/D converter
unsigned char focus_AD;             // Value desired for focus A/D converter
unsigned char zoom_min;             // Measured minimum value the pot can reach
unsigned char zoom_max;             // Measured maximum value the pot can reach
unsigned char focus_min;            // Measured minimum value the pot can reach
unsigned char focus_max;            // Measured maximum value the pot can reach
int zoom_digital;                 // Value desired for digital zoom
int zoom_estimate;                // Calculated digital zoom value
bit zoom_in_process;               // True if digital zooming is in process

// Transferred here from RICISR 1ms interrupt routine
bit checking_zoom;
unsigned char zooming;
unsigned char focusing;
unsigned char old_zoom_status;
unsigned char old_focus_status;

/*********************************************
* CODE TABLES
*****
****/
```

```

/*********************************************
*
* Name:      Main Loop
*
*          main()
*
* Description:
*
* This is the main loop of the RIC software. The main purpose for
* this routine is to monitor the Camera/IP links for activity then
* act on it. Activity on the Camera/IP links is recognized by the
* in_use/Full status in the Camera/IP receive buffers.
*
* This routine is also responsible for starting the wake up
* initialization and for zooming/focusing.
*
* Returns:    nothing, control does not return from this routine
*
****/
```

```

void main( void )
{
    // Now check if we have a valid application flash checksum or not
    if (startup_flags == POWER_UP)
    {
        if (check_flash(APP_CODE))
        {
            // Pass the boot software/firmware version on to the application code
            boot_SW_version = SOFTWARE_VER;
            startup_flags = APP_GOOD;
        }
    }
}
```

```

        jump_to_app();
    }
    else
        {startup_flags = INVALID_APP;}
}
init_sys();
if (startup_flags & BEGIN_ERASE)
{
    // Commanded from application side to do flash program (boot or app).
    // Erase sector specified by startup_flags and send ACK.
    if (erase_flash(startup_flags & ERASE_MASK)) send_ack();
    else    flag_error(BAD_FLASH_ERASE);
}

else
{
    if (startup_flags == INVALID_APP)
    {
        flag_error(BAD_APPLICATION_BLOCK);
    }
    calibrate();
}
// EX6 = 1;      // Enable External 6 interrupt for use in case of app code failure

while (TRUE)           // Infinite loop
{
    EAL = 1;           // interrupts are enabled

    // Are there any messages from the Host/Camera serial interfaces?
    if (HOST_Rxmsg_buf.in_use)
        process_host_msg();           /* A Host message has been collected */
    if (CAMERA_Rxmsg_buf.in_use)
        process_camera_msg();        /* A Camera message has been collected */

    // Are there any messages to send to the Host/Camera serial interfaces?
    if (CAMERA_Txmsg_status == TXMSG_COMPLETE && CAMERA_Txmsg_buf.in_use)
        start_camera_msg();
    if (HOST_Txmsg_status == TXMSG_COMPLETE && HOST_Txmsg_buf.in_use)
        start_host_msg();

    // Now check if too much time has passed during Host message reception, reset if so
    if (HOST_Rx_timeout)
    {
        // Get all the pointers and flags set to the default Rx state
        EAL = 0;
        HOST_Rxmsg_byte_ptr = &(HOST_Rxmsg_buf.Rxmsg_header.start_char);
        HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
        HOST_Rxmsg_byte_cnt = 0;
        HOST_Rxmsg_checksum = 0;
        no_HOST_Rx_timer = 0;
        HOST_Rx_timeout = FALSE;
        EAL = 1;
        flag_error(HOST_RX_TIMEOUT);
    }

    // Compare A/D values with desired values and adjust motors to correct.
    // ADTOL is used as a tolerance to allow motors time to stop.
    if ((zoom_status>zoom_AD)&&((zoom_status - zoom_AD) >ADTOL) )
        P5=0x27&(P5|0xFE);           // Zoom -
    else if ((zoom_AD>zoom_status)&&((zoom_AD-zoom_status)>ADTOL) )
        P5=0x3B&(P5|0xFE);           // Zoom +
    else if ((focus_status>focus_AD)&&((focus_status-focus_AD)>ADTOL))
        P5=0x0F&(P5|0xFE);           // Focus -
    else if ((focus_AD>focus_status)&&((focus_AD-focus_status)>ADTOL))
        P5=0x6B&(P5|0xFE);           // Focus +
    else P5=0x2B & (P5 | 0xFE);           // Zoom/Focus stop
}
}

```

```

*****
* Name:      Initialize Global Variables and Buffers
*
*           init_globals();
*
* Description:
*
*   This routine initializes global variables and buffers.
*
* Returns:    nothing
*
*****
void init_globals( void )
{
    HOST_Rxmsg_buf.in_use = FALSE;
    HOST_Txmsg_buf.in_use = FALSE;
    CAMERA_Rxmsg_buf.in_use = FALSE;
    CAMERA_Txmsg_buf.in_use = FALSE;
    HOST_ACK_timeout = FALSE;
    CAMERA_ACK_timeout = FALSE;
    HOST_Rx_timeout = FALSE;
    CAMERA_Rx_timeout = FALSE;
    HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
    CAMERA_Rxmsg_status = WAITING_FOR_START_CHAR;
    HOST_Txmsg_status = TXMSG_COMPLETE;
    CAMERA_Txmsg_status = TXMSG_COMPLETE;
    zoom_in_process = FALSE;
    focusing=0;
    zooming=0;
    zoom_cmd=0;
    focus_cmd=0;
    zoom_estimate=0;
    sending_ACK = FALSE;

    HOST_Rxmsg_byte_ptr = (unsigned char iodata *)&(HOST_Rxmsg_buf.Rxmsg_header.start_char);

    HOST_Txmsg_buf.Txmsg_header.start_char = START_CHAR;
}

```

```

*****
* Name:      Initialize System
*
*           init_sys();
*
* Description:
*
*   This routine is called on power up or boot.  This routine initializes
*   ports, variables, etc.
*
* Returns:    nothing
*
*****
void init_sys( void )
{
    init_globals();                                // initialize global variables and buffers

    // Initialize internal timers 0/1
    TR0 = 0;                                     // disable timer 0
    TMOD = TMR1_CNTRL_REG | TMR0_CNTRL_REG;      // Timer 0 is 16-bit, timer 1 is 8-bit auto-reload
    TH0 = 0xF8;                                    // F8D8 = 1988 us
    TL0 = 0xD8;
    TR0 = 1;                                     // enable timer0
    TR1 = 0;                                     // Disable timer 1 (unused)
    ET1 = 0;                                     // disable timer 1 interrupt

    // Now initialize the IP connection (ser port 0) (might be power on with bad app)

```

```

RENO = 0;           // disable serial port 0 receiver
SM0 = 0;           // serial port 0 is 8-bit
SM1 = 1;           // "
BD = 1;           // serial port 0 baud rate derived from Baud Rate Generator
PCON |= 0x80;     // Set SMOD bit so serial port 0 baud rate is doubled
SORELH = 0xFF;    // 3DC = 9600 baud (top 5 bits don't count)
SORELL = 0xDC;    // "
RENO = 1;           // enable serial port 0 receiver

// Initialize the camera connection (ser port 1)
S1CON = 0x80;    // set 8-bit mode and disable receiver
S1RELH = 0xFF;    // 3DC = 9600 baud (top 5 bits don't count)
S1RELL = 0xDC;    // "
S1CON = 0xB0;    // enable receiver

IEN2 &= 0xFE;      // Disable camera serial port 1 interrupt
ES0 = 1;           // Enable IP serial port 0 interrupt
IEN2 |= 0x01;      // Enable camera serial port 1 interrupt

// iris_setting=IRIS_DEFAULT;    // Initialize iris gain
set_iris(iris_setting);
P5 = 0x2B;          // Turn on camera, turn off zoom and focus
latch_status = 0x02; // Set latch status to camera on, iris automatic

// Initialize A/D converter
ADM = 0;           // Single conversion mode
ADEX = 0;           // No external trigger
EADC = 0;           // ADC interrupt disabled
ADCON1 = 0x80;     // Set MUX to zoom input & /8 clock
ADDATL = 0;          // Start conversion
while (BSY) {};    // Wait until conversion complete
zoom_AD = ADDATH; // Set zoom to current value
zoom_status = zoom_AD; // "
zoom_cmd = (zoom_AD<<3); // "
zoom_digital = 0x30; // Set digital zoom to 1X
ADCON1 = 0x81;     // Set MUX to focus input & /8 clock
ADDATL = 0;          // Start conversion
while (BSY) {};    // Wait until conversion complete
focus_AD = ADDATH; // Set focus to current value
focus_status = focus_AD; // "
focus_cmd=(focus_AD<<4); // "

// Setup interrupts and disable some interrupts used in application code
EX0 = 0;           // Disable External 0 interrupt
EX1 = 0;           // Disable External 1 interrupt
EX2 = 0;           // Disable External 2 interrupt
EX3 = 0;           // Disable External 3 interrupt
EX4 = 0;           // Disable External 4 interrupt
EX5 = 0;           // Disable External 5 interrupt
ET0 = 1;           // Enable Timer 0 (Msec tick) interrupt
IP0 |= 2;          // Make Msec (Timer 0) highest priority
IP1 |= 2;          // "
EAL = 1;           // Enable ALL setup interrupts
}

/*********************************************
*****
void calibrate(void)
{
    zoom_status=0x7F;      // Set status to arbitrary center (just to start)
    zoom_AD=0xFF;          // Set command to 0xFF (the maximum)
    P5=0x3B&(P5|0xFE);    // Start zoom +
    while(zoom_AD^zoom_status) {} // Wait for timer to stop zoom
    zoom_max=zoom_status; // Save value

    zoom_status=0x7F;      // Set status to arbitrary center (just to start)
    zoom_AD=0x00;          // Set command to zero (the minimum)
    P5=0x27&(P5|0xFE);    // Start zoom -
    while(zoom_AD^zoom_status) {} // Wait for timer to stop zoom
    zoom_min=zoom_status; // Save value
}

```

```

focus_status=0x7F;      // Set status to arbitrary center (just to start)
focus_AD=0xFF;          // Set command to 0xFF (the maximum)
P5=0x6B&(P5|0xFE);    // Start focus +
while(focus_AD^focus_status) {} // Wait for timer to stop focus
focus_max=focus_status; // Save value

focus_status=0x7F;      // Set status to arbitrary center (just to start)
focus_AD=0x00;          // Set command to zero (the minimum)
P5=0x0F&(P5|0xFE);    // Start focus -
while(focus_AD^focus_status) {} // Wait for timer to stop focus
focus_min=focus_status; // Save value

// If max isn't greater than minimum use default values
if(zoom_max<=zoom_min)
{
    zoom_max=0xF0;
    zoom_min=0x0F;
}
if(focus_max<=focus_min)
{
    focus_max=0xF0;
    focus_min=0x0F;
}

else P5=0x2B & (P5 | 0xFE); // Zoom/Focus stop
}

/*****
*
* Name:      flag_error(error type)
*
* Description:
*
*   Common calling point for system errors as they occur
*
* Returns:    nothing
*
*****
void flag_error(unsigned char error_type) reentrant
{
    unsigned char sending_host_error_msg = FALSE;
    unsigned char error_type_parm;
    int ii;
    int junk;

    /* Now determine error and action needed */
    switch (error_type)
    {
        case HOST_Rx_TIMEOUT:
            sending_host_error_msg = TRUE;
            error_type_parm = HOST_RX_TIMEOUT_ERROR; // 0x34
            break;
        case BAD_HOST_Rx_MSG_CHECKSUM:
            sending_host_error_msg = TRUE;
            error_type_parm = 0x39;
            break;
        case BAD_APPLICATION_BLOCK:
            sending_host_error_msg = TRUE;
            error_type_parm = APP_BLOCK_BAD; // 0x31
            break;
        case BAD_FLASH_ERASE:
            sending_host_error_msg = TRUE;
            error_type_parm = FLASH_ERASE_ERROR; // 0x32
            break;
        case BAD_FLASH_PROGRAMMING:
            sending_host_error_msg = TRUE;
            error_type_parm = FLASH_PROGRAM_ERROR; // 0x33
            break;
        case ERROR_4:
            sending_host_error_msg = TRUE;
}

```

```

        error_type_parm = 0x34;           // 0x34
        break;
    default:
        break;
    }

    // Now check if an error msg needs to be sent to host
    if (sending_host_error_msg)
    {
        ii=2000;
        while (HOST_Txmsg_buf.in_use)           // Wait for buffer to empty or ~50ms
        {
            junk = P5;                      // Time wasting commands to use ~50ms
            if (!(ii--)) break;
        }

        // Now send "Err" and the error number
        HOST_Txmsg_buf.msg_length=4;
        HOST_Txmsg_buf.Txmsg_header.start_char='E';
        HOST_Txmsg_buf.Txmsg_header.dest_id='r';
        HOST_Txmsg_buf.Txmsg_header.msg_type='r';
        HOST_Txmsg_buf.msg_data[0]= error_type_parm;
        HOST_Txmsg_buf.in_use=TRUE;
    }
}

*****ACK*****
*****ACK*****


void send_ack(void) // Just send ACK and hope it doesn't interrupt anything
{
    ESO = 0;                      // Disable IP serial port 0 interrupt
    sending_ACK = TRUE;           // Set sending_ACK flag
    S0BUF=ACK;
    ESO = 1;                      // Enable IP serial port 0 interrupt
}

*****This processes messages from the Camera. Only zoom status is expected and when it is
* received, the digital zoom is updated.
*****


void process_camera_msg(void)
{} // Since zoom status doesn't work right on the camera, this is only a placeholder.
/* unsigned char msg_type;
   unsigned char idata *msg_ptr;

   msg_type = CAMERA_Rxmsg_buf.Rxmsg_header.cmd;
   msg_ptr = &CAMERA_Rxmsg_buf.msg_data[0];

   switch (msg_type)
   {
       case 'Z': // This message is Zoom Status Response from Camera
           if (zoom_digital>*msg_ptr) // Measured values are compared with
           {                           // desired and adjustments made.
               zoom_in();
               zoom_in_process=TRUE;
           }
           if (zoom_digital<*msg_ptr)
           {
               zoom_out();
               zoom_in_process=TRUE;
           }
           if (!(zoom_digital ^ *msg_ptr))
           {
               zoom_stop();
               zoom_in_process=FALSE;
           }
       break;

   default:

```

```

        break;
    }
}

void set_iris(unsigned char iris)      // This sets Average Picture Level (APL) in lieu of
{                                     // setting iris.
    if (iris>99) iris=99; // check limits
    if (iris<0) iris=0;

    CAMERA_Txmsg_buf.msg_length=6;                                // Set up command to send to camera
    CAMERA_Txmsg_buf.Txmsg_header.start_char=0x02;
    CAMERA_Txmsg_buf.Txmsg_header.dest_id=0x01;
    CAMERA_Txmsg_buf.Txmsg_header.msg_type='c';
    CAMERA_Txmsg_buf.msg_data[0]='L';
    CAMERA_Txmsg_buf.msg_data[1]= (iris/10) + '0';                // Convert to ASCII, MSB first.
    CAMERA_Txmsg_buf.msg_data[2]= (iris%10) + '0';
    CAMERA_Txmsg_buf.in_use=TRUE;
}

void send_latch_status(void) // Send iris & power status
{
    while (sending_ACK){}; // Wait for ACK to finish
    HOST_Txmsg_buf.msg_length=6;
    HOST_Txmsg_buf.Txmsg_header.start_char=0xF8;
    HOST_Txmsg_buf.Txmsg_header.dest_id=0x01;
    HOST_Txmsg_buf.Txmsg_header.msg_type='L';
    HOST_Txmsg_buf.msg_data[0]=latch_status;
    HOST_Txmsg_buf.msg_data[1]=latch_status;
    HOST_Txmsg_buf.msg_data[2]=latch_status;
    HOST_Txmsg_buf.in_use=TRUE;
}

//*********************************************************************
* Lens status includes zoom and focus. Focus is 12 bits to GDFS but
* only 8 bits to the camera, since we are only using the 8 MSB of the
* A/D converter.
* The zoom is also 12 bits to GDFS but only 8 bits to the A/D and 11
* bits for digital zoom. Since the camera digital zoom setting can't
* be determined due to a camera internal software bug, we will just
* report the last zoom command received.
*****//********************************************************************

void send_lens_status(void) // Send lens status to GDFS
{
    while (sending_ACK){}; // Wait for ACK to finish
    HOST_Txmsg_buf.msg_length=9;
    HOST_Txmsg_buf.Txmsg_header.start_char=0xF8;
    HOST_Txmsg_buf.Txmsg_header.dest_id=0x01;
    HOST_Txmsg_buf.Txmsg_header.msg_type='v';
    HOST_Txmsg_buf.msg_data[0]=((zoom_cmd&0xF00)>>8) | 0x30;
    HOST_Txmsg_buf.msg_data[1]=((zoom_cmd&0x0F0)>>4) | 0x30;
    HOST_Txmsg_buf.msg_data[2]=(zoom_cmd&0x00F) | 0x30;
    HOST_Txmsg_buf.msg_data[3]=((focus_status&0xF0)>>4) | 0x30;
    HOST_Txmsg_buf.msg_data[4]=((focus_status&0x0F)) | 0x30;
    HOST_Txmsg_buf.msg_data[5]= 0x30;
    HOST_Txmsg_buf.in_use=TRUE;
}

void request_lens_status(void) // Request digital zoom status from camera
{
    CAMERA_Txmsg_buf.msg_length=5;
    CAMERA_Txmsg_buf.Txmsg_header.start_char=0x02;
    CAMERA_Txmsg_buf.Txmsg_header.dest_id=0x01;
    CAMERA_Txmsg_buf.Txmsg_header.msg_type='c';
    CAMERA_Txmsg_buf.msg_data[0]='Z';
    CAMERA_Txmsg_buf.msg_data[1]='S';
    CAMERA_Txmsg_buf.in_use=TRUE;
}

void zoom_in(void) // Start camera zooming in
{
    CAMERA_Txmsg_buf.msg_length=5;
}

```

```

CAMERA_Txmsg_buf.Txmsg_header.start_char=0x02;
CAMERA_Txmsg_buf.Txmsg_header.dest_id=0x01;
CAMERA_Txmsg_buf.Txmsg_header.msg_type='c';
CAMERA_Txmsg_buf.msg_data[0]='Z';
CAMERA_Txmsg_buf.msg_data[1]='i';
CAMERA_Txmsg_buf.in_use=TRUE;
}

void zoom_out(void)           // Start camera zooming out
{
    CAMERA_Txmsg_buf.msg_length=5;
    CAMERA_Txmsg_buf.Txmsg_header.start_char=0x02;
    CAMERA_Txmsg_buf.Txmsg_header.dest_id=0x01;
    CAMERA_Txmsg_buf.Txmsg_header.msg_type='c';
    CAMERA_Txmsg_buf.msg_data[0]='Z';
    CAMERA_Txmsg_buf.msg_data[1]='o';
    CAMERA_Txmsg_buf.in_use=TRUE;
}

void zoom_stop(void)          // Stop camera zooming
{
    CAMERA_Txmsg_buf.msg_length=5;
    CAMERA_Txmsg_buf.Txmsg_header.start_char=0x02;
    CAMERA_Txmsg_buf.Txmsg_header.dest_id=0x01;
    CAMERA_Txmsg_buf.Txmsg_header.msg_type='c';
    CAMERA_Txmsg_buf.msg_data[0]='Z';
    CAMERA_Txmsg_buf.msg_data[1]='s';
    CAMERA_Txmsg_buf.in_use=TRUE;
}

```

```

*****
* Name:      FLASH.C
*
* Functions:  flash_cmd();
*              ascii_to_byte();
*              amd_program_buffer();
*              program_buffer();
*              program_checksum();
*              amd_erase();
*              erase_flash();
*              check_flash();
*
* Description:
*
*     Routines to handle the IP (Host) <-> Flash programming
*
* History: Adapted for VIS 12/2/98
*
*****
```

```

#include "ric.h"           /* Constants, hardware definitions, etc. */
#include "proto.h"          /* Function prototypes for all modules */
#include "ricglob.h"         /* Global variable (externs) definitions */

/*
 * FLASH_CMD writes the CMD to the Flash chip after writing to the data
 * registers in the appropriate sequence.
 */
void flash_cmd(unsigned char cmd)
{
    P3_latch_image = P3;           // save A16
    A16_NOT = 1;                  // clear A16
    DATA_REG0 = 0xAA;
    DATA_REG1 = 0x55;
    DATA_REG0 = cmd;
    A16_NOT = P3_latch_image & 0x10; // restore A16
}

/*
 * ascii_to_byte converts 1 ascii character to a hex byte/nibble
 */
unsigned char ascii_to_byte(unsigned char ascii_byte)
{
    // Convert ascii character to hex byte
    if (ascii_byte >= '0' && ascii_byte <= '9')
        ascii_byte -= '0';
    else
    {
        ascii_byte -= 'A';
        ascii_byte += 10;
    }
    return (ascii_byte);
}

/*
 * AMD_PROGRAM_BUFFER attempts to program LEN bytes into an AMD Flash memory at
 * the BASE_ADDRESS passed from the BUF passed. This function returns the
 * success or failure of the programming.
 */
bit amd_program_buffer(unsigned int base_address, unsigned char idata *buf_ptr, unsigned char
len)
{
    unsigned char flash_byte;
    unsigned char val;

    /*
     * Program each byte from BUF into Flash memory.
     */

```

```

while (len-- != 0)
{
    /*
     * Write the FLASH_PROGRAM command to the generic command register and
     * then the byte to be programmed at the appropriate address.
     */
    flash_cmd(FLASH_PROGRAM);
    flash_byte = ascii_to_byte(*buf_ptr++) * 0x10;
    flash_byte += ascii_to_byte(*buf_ptr++);
    XBYTE[base_address] = flash_byte;

    /*
     * Wait for programming of this byte to complete. Programming is
     * complete when the DATA_POLLING bit of the byte programmed into Flash
     * is equal to the corresponding bit of the byte from the buffer.
     * The Flash chip complements this bit while programming is in
     * progress. Programming failed if the TIMEOUT bit is ever set.
     */
    do
    {
        val = XBYTE[base_address];
    } while (((val & DATA_POLLING) != (flash_byte & DATA_POLLING)) &&
        !(val & TIMEOUT));

    /*
     * If the data readback does not match the data programmed then Flash
     * programming has failed.
     */
    if (val != flash_byte)
        return FALSE;

    /*
     * Increment BASE_ADDRESS to the next address in Flash.
     */
    ++base_address;
}

/*
 * All bytes in the buffer were programmed successfully.
 */
return TRUE;
}

/*
 * PROGRAM_BUFFER is the generic entry point for programming a buffer into
 * Flash memory. Upon entry, BUF points to a buffer whose first two bytes
 * contain the HEX ASCII byte count in buffer followed by the HEX ASCII
 * address for programming the rest of the data in the buffer (4 bytes)
 * This returns the success or failure status of programming BUF into
 * Flash. (0 = BAD)
 */
bit program_buffer(unsigned char idata *buf_ptr)
{
    unsigned char len;
    unsigned int address;

    len = ascii_to_byte(*buf_ptr++) * 0x10;
    len += ascii_to_byte(*buf_ptr++);
    address = ascii_to_byte(*buf_ptr++) * 0x1000;
    address += ascii_to_byte(*buf_ptr++) * 0x100;
    address += ascii_to_byte(*buf_ptr++) * 0x10;
    address += ascii_to_byte(*buf_ptr++);

    // Now check the sanity of the address and/or length
    if (len <= 0x10 && address < 0x7FF0)
    {
        // See if this record is a repeat from last time (RF retry)
        if (last_dnload_msg_state == address)
            return (TRUE);
        else

```

```

        {
            last_dnload_msg_state = address;
            return amd_program_buffer(address, buf_ptr + 2, len);
        }
    }
else
    return (FALSE);                                // Proper error msg will be generated on
return
}

/*
 * PROGRAM_CHECKSUM programs a checksum into final two bytes of the block(s)
 * used in Flash memory. This function returns the success or failure of
 * the programming operation.
 */
bit program_checksum(void)
{
    unsigned char val;
    unsigned int cnt;
    unsigned int end;
    unsigned int xsum = 0;

    /*
     * Initialize the END address for this block as the BASE_ADDRESS plus the
     * LEN, less the CHECKSUM_SIZE bytes at the end of the block comprising the
     * checksum offset.
     */
    if (prog_block == APP_CODE)
        end = APP_SIZE - CHECKSUM_SIZE;
    else
        end = BOOT_SIZE - CHECKSUM_SIZE;

    for (cnt = 0; cnt < end; ++cnt)
        xsum += XBYTE[cnt];

    xsum = FLASH_CHECKSUM - xsum;

    /*
     * Program the checksum into Flash.
     */
    flash_cmd(FLASH_PROGRAM);
    XBYTE[end] = (unsigned char)(xsum >> 8);

    do                                // Program the 1st byte and check flash data programming results
    {
        val = XBYTE[end];
    } while (((val & DATA_POLLING) != ((unsigned char)(xsum >> 8) & DATA_POLLING)) &&
            !(val & TIMEOUT));
    if (val != (unsigned char)(xsum >> 8))
        return FALSE;
    ++end;

    flash_cmd(FLASH_PROGRAM);
    XBYTE[end] = (unsigned char)xsum;

    do                                // Program the 2nd byte and check flash data programming results
    {
        val = XBYTE[end];
    } while (((val & DATA_POLLING) != ((unsigned char)xsum & DATA_POLLING)) &&
            !(val & TIMEOUT));
    if (val != (unsigned char)xsum)
        return FALSE;
    else
        return TRUE;
}

/*
 * AMD_ERASE erases the 16K block of Flash memory containing the SECTOR_ADDRESS
 * passed. This function returns TRUE if the sector is erased successfully and
 * FALSE if an error occurs. Note that the SECTOR_ADDRESS passed is a 16 bit

```

```

* address.  The 17th address bit (these are 29F010's) must already be set
* appropriately.
*/
bit amd_erase(unsigned int sector_address)
{
    unsigned char val;

    /*
     * Enable erase operations and then issue the erase command.  Timing is
     * critical (the erase operation begins 80us after the last write) so we
     * must not be interrupted.
     */
    EAL = 0;
    flash_cmd(ERASE_ENABLE);
    P3_latch_image = P3;                      // save A16
    A16_NOT = 1;                             // clear A16
    DATA_REG0 = 0xAA;
    DATA_REG1 = 0x55;
    A16_NOT = P3_latch_image & 0x10;        // restore A16
    XBYTE[sector_address] = SECTOR_ERASE;
    EAL = 1;

    /*
     * Erase is complete when the DATA_POLLING bit is set or the TIMEOUT bit is
     * set.
     */
    do
    {
        val = XBYTE[sector_address];
    } while ((val & (DATA_POLLING | TIMEOUT)) == 0);

    flash_cmd(FLASH_RESET);

    /*
     * If we exited the above loop because the DATA_POLLING bit is set, return
     * TRUE (erase was successful).  Otherwise, the TIMEOUT bit must be set.
     * In that case erase failed and we will return FALSE.
     */
    return val & DATA_POLLING;
}

/*
 * ERASE_FLASH erases the Flash memory BLOCK passed as required.  Upon exit,
 * the global FLASH_ERASED flag is set according to the success or failure of
 * the operation.
*/
bit erase_flash(unsigned char block)
{
    bit flash_erased = TRUE;
    unsigned char segments;
    unsigned int address;

    /*
     * Set the global PROG_BLOCK to the BLOCK passed.  Compute the ADDRESS and
     * number of SEGMENTS from the FLASH_TABLE.
     */
    prog_block = block;
    address = 0;

    /*
     * Set FLASH_A16 according to the BLOCK.
     */
    if (block == APP_CODE)
    {
        A16_NOT = 0;                      // set A16 high
        P3_latch_image = P3;            // save image if P3 (just in case)
        segments = 2;
    }
    else
    {
        A16_NOT = 1;                      // set A16 low
    }
}

```

```

        P3_latch_image = P3;      // save image if P3 (just in case)
        segments = 1;
    }

/*
 * Erase each of the SEGMENT_SIZE SEGMENTS, cumulatively anding the
 * FLASH_ERASED result.
 */
while (segments--)
{
    flash_erased &= amd_erase(address);
    address += SEGMENT_SIZE;
}
last_dnload_msg_state = 1;
return (flash_erased);
}

/*
 * CHECK_FLASH returns a boolean indicating whether the Flash memory BLOCK
 * passed is valid. A valid block of Flash memory will sum to the
 * FLASH_CHECKSUM value. Upon entry into this routine, we are running in RAM.
 */
bit check_flash(unsigned char block)
{
    unsigned int cnt;
    unsigned int end;
    unsigned int xsum = 0;

    /*
     * If the BLOCK to be checked is the APP_CODE, set A16 high (FLASH_A16 low)
     * and read the Flash as external data bytes.
     */
    if (block == APP_CODE)
    {
        A16_NOT = 0;           // set A16 high
        P3_latch_image = P3;  // save image of P3 (just in case)
        end = APP_SIZE - CHECKSUM_SIZE;
    }
    else
    {
        A16_NOT = 1;           // set A16 low
        P3_latch_image = P3;  // save image of P3 (just in case)
        end = BOOT_SIZE - CHECKSUM_SIZE;
    }
    for (cnt = 0; cnt < end; ++cnt)
        xsum += XBYTE[cnt];
    /*
     * Add the last two bytes of Flash as a word value.
     */
    xsum += XWORD[cnt / 2];
    return xsum == FLASH_CHECKSUM;
}
}

```

```
*****
*                                         *
* Name:      HOSTMSG.C                 *
*                                         *
* Functions: process_host_msg();        *
*             xmit_host_msg();          *
*             start_host_msg();         *
*             set_carrier_frequency();   *
*                                         *
* Description:                         *
*                                         *
*     Routines to handle the IP (Host) <-> board (RIC) interface messages *
*                                         *
* History:  Adapted for VIS 12/2/98  *
*                                         *
*****
```

```
#include "ric.h"           /* Constants, hardware definitions, etc. */
#include "proto.h"          /* Function prototypes for all modules */
#include "ricglob.h"         /* Global variable (externs) definitions */
```

```
*****
* GLOBAL VARIABLES
*****
```

```
// This buffer is declared here to link in before RIC.C declarations
tx_msg_header HOST_Txmsg_header; // Header building area for Host msgs
```

```
*****
*                                         *
* Name:      Process host Messages      *
*                                         *
* Description:                         *
*                                         *
*     This routine processes host messages after the message has been      *
*     completely collected.  If messages are intended for this unit to      *
*     act on the message this routine will initiate the action.  If      *
*     the message is being directed to another unit, this routine will      *
*     initiate that action.          *
*                                         *
* Returns:    nothing
*                                         *
*****
```

```
void process_host_msg(void)
{
    bit sending_host_other_msg = FALSE;
    unsigned char msg_dest_id;
    unsigned char msg_type;
    unsigned char data_len;
    unsigned char iodata *msg_ptr;
    int zoom_setting;           // Desired value of zoom from GDFS
    int focus_setting;          // Desired value of focus from GDFS

    msg_dest_id = HOST_Rxmsg_buf.Rxmsg_header.id;
    msg_type = HOST_Rxmsg_buf.Rxmsg_header.cmd;
    data_len = HOST_Rxmsg_buf.Rxmsg_header.data_length;
    msg_ptr = &HOST_Rxmsg_buf.msg_data[0];

    /* What kind of message is it? */
    switch(msg_type)
    {
        case 'L':
            send_ack();
            switch(*msg_ptr)
            {
                case 'P': // Toggle camera power
                    P5 ^= 0x01;
                    if (P5&0x01) latch_status |= 0x02;
                    else latch_status &= 0xFD;
            }
    }
}
```

```

        send_latch_status();
        break;
    case 'M':           // Toggle iris control (set to default)
        iris_setting=IRIS_DEFAULT;
        set_iris(iris_setting);
        latch_status ^= 0x01;
        send_latch_status();
        break;
    case '?':           // Send iris & power status
        send_latch_status();
        break;
    }
    break;
case 'v':           // Set zoom & focus (see ric.c: set_lens_status)
    send_ack();
    // zoom_setting is an intermediate working variable
    // zoom_cmd is the decoded 12 bit GDFS zoom command
    // zoom_AD is the A/D command
    // zoom_digital is the digital zoom command
    zoom_setting=*msg_ptr;
    zoom_setting=(zoom_setting&0x0F)<<8;
    msg_ptr++;
    zoom_setting=zoom_setting+((*msg_ptr&0x0F)<<4);
    msg_ptr++;
    zoom_setting=zoom_setting+(*msg_ptr&0x0F);
    msg_ptr++;
    if (zoom_setting<0x7F8)
    {
        zoom_AD=((zoom_setting>>3)*(zoom_max-zoom_min))/0xFF +zoom_min;
        if (zoom_AD>zoom_max) zoom_AD=zoom_max;
        zoom_digital=0x0;
        if (zoom_cmd>0x7F8)
            zoom_estimate=2500; // Make sure digital zoom goes to min
    }
    else
    {
        if (zoom_cmd<0x7F8) zoom_AD=zoom_max; // Only max out the A/D
        zoom_digital=(zoom_setting-0x7F7); // if prev. cmd didn't.
        if (zoom_digital>1700)
        {
            zoom_digital=1700;
            zoom_estimate=0; // Make digital zoom go to max
        }
        if (zoom_digital<0) zoom_digital=0; // Just in case of error
    }
    zoom_cmd=zoom_setting; // This saves the 12 bit zoom command
    if (zoom_digital==zoom_estimate)
    {
        digital_zoom_in_timer=0;
        digital_zoom_out_timer=0;
        zoom_stop();
    }
    if (zoom_digital>zoom_estimate)
    {
        digital_zoom_in_timer=zoom_digital-zoom_estimate;
        digital_zoom_out_timer=0;
        zoom_in();
    }
    if (zoom_digital<zoom_estimate)
    {
        digital_zoom_out_timer=zoom_estimate-zoom_digital;
        digital_zoom_in_timer=0;
        zoom_out();
    }
    focus_setting=(*msg_ptr&0x0F)<<8;
    msg_ptr++;
    focus_setting=focus_setting+((*msg_ptr&0x0F)<<4);
    msg_ptr++;
    focus_setting=focus_setting+(*msg_ptr&0x0F);
    msg_ptr++;

```

```

focus_cmd=focus_setting;
focus_AD=((focus_setting>>4)*(focus_max-focus_min))/0xFF +focus_min;
if (focus_AD>focus_max) focus_AD=focus_max;
break;
case 'I':
    send_ack();
    switch(*msg_ptr)
    {
        case 'C':      // Close iris
            if(iris_setting<IRIS_STEP)
                iris_setting=IRIS_STEP;
            iris_setting=iris_setting-IRIS_STEP;
            set_iris(iris_setting);
            latch_status |= 0x01;
            break;
        case 'O':      // Open iris
            if(iris_setting>(99-IRIS_STEP))
                iris_setting=(99-IRIS_STEP);
            iris_setting=iris_setting+IRIS_STEP;
            set_iris(iris_setting);
            latch_status |= 0x01;
            break;
        case 'S':      // Stop iris command (ignore this)
            break;
    }
    break;
case 'V':          // Assume next char is "?", get and send lens status
    send_ack();    // Although the sent lens status will not reflect the
    request_lens_status(); // status requested, it will be there for
    send_lens_status(); // the next update.
    break;
case 'P':          // This is a "pass-byte-to-camera" for future use
    send_ack();
    S1BUF = *msg_ptr;
    break;

case BEGIN_DOWNLOAD:
    /* BEGIN_DOWNLOAD initiates the Flash erase procedure. Erase the
     * block of Flash memory indicated in the message (0 = boot, 1 =
     * application). Before erasing, a return to boot sector and RAM
     * is done to ensure Flash erase isn't attempted while running in
     * Flash. */
    startup_flags = BEGIN_ERASE|(*msg_ptr);      // Set flags to erase
    _return_to_boot();                            // Return to RAM boot sector
    break;

case INTEL_HEX_RECORD:
    /* Program the Flash memory with the data passed in the buffer.
     * If the status of this programming operation is bad (0), send
     * error msg. If good, echo header portion only of this msg */
    if (!program_buffer(msg_ptr + 1))      // This was +2
    {
        sending_host_other_msg = TRUE;
        flag_error(BAD_FLASH_PROGRAMMING);
    }
    else
    {
        send_ack();
    }
    break;
case END_DOWNLOAD:
    /* The PROG_DONE command causes the checksum to be programmed into
     * Flash. If the status of this programming operation is bad (0),
     * send error msg. If good, send ACK */
    if (!program_checksum())
    {
        sending_host_other_msg = TRUE;
        flag_error(BAD_FLASH_PROGRAMMING);
        last_dnload_msg_state = 0;
    }
}

```

```

        else
        {
            send_ack();
            while(sending_ACK) {};
            reboot();
        }
        break;
    case REBOOT:
        // Do a Warm boot (goes back to boot code)
        reboot();
        break;
    default:
        break;
    }

    HOST_Rxmsg_buf.in_use = FALSE;
}

//****************************************************************************
/*
 * Name:          Start Host/Camera Message
 *
 * start_host_msg(void);
 * start_camera_msg(void);
 *
 * Description:
 *
 * This routine starts sending the message from Tx buffer to Uart
 *
 * Returns:      Nothing
 */
//****************************************************************************
/* To use this code to send a message:
   1. Load HOST_Txmsg_buf.msg_length with length of message including
   start characters but not checksum.
   2. Load HOST_Txmsg_buf.Txmsg_header.start_char with 0xF8.
   3. "           "           .dest_id with 0x01.
   4. "           "           .msg_type with {type}.
   5. "           "           .msg_data with {all other bytes of data}.
Use CAMERA_Txmsg... instead of HOST_Txmsg if sending to camera. Any 4 bytes
can be used instead of the 0xF8, 0x01, {type} and {length of data}.
   6. Call start_host_msg to send to GDFS, or start_camera_msg to send
   to camera.
*/
void start_host_msg(void)
{
    // Get all the pointers and counters set up
    HOST_Txmsg_byte_cnt = HOST_Txmsg_buf.msg_length;
    HOST_Txmsg_byte_ptr = (unsigned char idata *)&(HOST_Txmsg_buf.Txmsg_header.start_char);
    EAL = 0;                                // Disable all interrupts
    HOST_Txmsg_status = SENDING_DATA;
    HOST_Txmsg_checksum = 0;
    S0BUF = *HOST_Txmsg_byte_ptr++;
    HOST_Txmsg_byte_cnt--;
    EAL = 1;                                // Enable all interrupts
}
void start_camera_msg(void)
{
    // Get all the pointers and counters set up
    CAMERA_Txmsg_byte_cnt = CAMERA_Txmsg_buf.msg_length;
    CAMERA_Txmsg_byte_ptr = (unsigned char idata
*) &(CAMERA_Txmsg_buf.Txmsg_header.start_char);
    EAL = 0;
    CAMERA_Txmsg_status = SENDING_DATA;
    CAMERA_Txmsg_checksum = 0;
    S1BUF = *CAMERA_Txmsg_byte_ptr++;
    CAMERA_Txmsg_byte_cnt--;
    EAL = 1;
}

```

```

*****
* Name: RICISR.C
*
* Functions: host_isr();
*             rf_isr();
*             Msec_tick_timer();
*
* Descriptions:
*
* These routines are Interrupt Service Routines and should not be
* called by any other routine.
*
* History: Adapted for VIS 12/2/98
*****
****

#include "ric.h"          /* Constants, hardware definitions, etc. */
#include "proto.h"         /* Function prototypes for all modules */
#include "ricglob.h"        /* Global variable (externs) definitions */

*****
* Name: Host Interrupt Service Routine (ISR)
*
* host_isr();
*
* Description:
*
* This ISR sends and receives character to and from the microcontroller
* UART of the host serial interface (Host-to-board).
*
* Returns: nothing
*****
void host_isr( void ) interrupt 4 using 1      // 0x23; ser port 0, GDFS
{
    unsigned char new_char;
    int msg_count;
    int data_length;      // length the message should be

    /*
     * Has a character been received?
     */
    if ( RIO )
    {
        RIO = 0;           // Clear interrupt pending condition

        new_char = S0BUF;

        switch (HOST_Rxmsg_status)
        {
            case WAITING_FOR_START_CHAR:
                HOST_Rxmsg_byte_ptr = &HOST_Rxmsg_buf.Rxmsg_header.start_char;
                HOST_Rxmsg_byte_cnt = 0;
                HOST_Rxmsg_checksum = 0;
                no_HOST_Rx_timer = 0;
                if (!(new_char ^ START_CHAR)) // If start character, go on
                {
                    S0BUF=new_char; // TEST ONLY ****
                    HOST_Rxmsg_status = RECEIVING_ID;
                    no_HOST_Rx_timer = 255;           // Set Rx timeout to 255
                }
                msecs
                *HOST_Rxmsg_byte_ptr++ = new_char; // Store byte
                HOST_Rxmsg_byte_cnt++;           // increment byte counter
                break;
            }
            if (new_char == ACK)
            {
                HOST_ACK_timer = 0; // If ACK, disable ACK timer
                HOST_ACK_timeout = FALSE;
            }
        }
    }
}

```

```

        }
        if (new_char == NACK)
            HOST_ACK_timeout = TRUE; // If NACK, flag ACK failure
        if ((new_char != ACK) && (new_char != NACK)
            && (new_char ^ START_CHAR)) // If other, test
        {
            S0BUF = new_char+1; // Echo bad character +1
            HOST_Rxmsg_status=TEST_MODE2;
            break;
        }
    */
    break;
    case RECEIVING_ID:
        if (new_char != ID) // If not ID, it must be length for
        {
            // reprogramming Flash
// S0BUF=0x01; // TEST ONLY ****
            if (new_char > 52) // simple sanity check
            {
                // if too high, start over
                HOST_Rxmsg_byte_ptr = &HOST_Rxmsg_buf.Rxmsg_header.start_char;
                HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
                HOST_Rxmsg_byte_cnt = 0;
                HOST_Rxmsg_checksum = 0;
                no_HOST_Rx_timer = 0;
                break;
            }
            else // new character must be length
            {
                HOST_Rxmsg_status = RECEIVING_EPROM_CMD;
                no_HOST_Rx_timer = 255; // Set Rx timeout to 255
msecs
                HOST_Rxmsg_byte_ptr++; // Skip ID field
                *HOST_Rxmsg_byte_ptr++ = new_char; // Store byte in length
                HOST_Rxmsg_byte_cnt++; // increment byte counter
                HOST_Rxmsg_checksum = new_char+0xF8; // checksum=start char+this
                data_length = new_char;
                break;
            }
        }
        HOST_Rxmsg_status = RECEIVING_CMD;
        no_HOST_Rx_timer = 255; // Set Rx timeout to 255 msecs
        *HOST_Rxmsg_byte_ptr++ = new_char; // Store byte
        HOST_Rxmsg_byte_cnt++; // increment byte counter
        HOST_Rxmsg_checksum ^= new_char; // XOR with checksum
        break;
    case RECEIVING_CMD:
        HOST_Rxmsg_status = RECEIVING_MESSAGE_DATA;
        no_HOST_Rx_timer = 255; // Set Rx timeout to 255 msecs
        switch (new_char)
        {
            case 'v':
                data_length =6;
                *HOST_Rxmsg_byte_ptr++ = data_length; //store length in data.length
                HOST_Rxmsg_byte_cnt++; // but don't include in chksum
                break;
            case 'L':
            case 'I':
            case 'V':
            case 'P':
            case 'X':
            case 'D':
                data_length =1;
                *HOST_Rxmsg_byte_ptr++ = data_length; //store length in data.length
                HOST_Rxmsg_byte_cnt++; // but don't include in chksum
                break; // Break from inner loop but fall to next case!
        default:
// S0BUF=new_char; // TEST ONLY ****
            HOST_Rxmsg_byte_ptr = &HOST_Rxmsg_buf.Rxmsg_header.start_char;
            HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
            HOST_Rxmsg_byte_cnt = 0;
            HOST_Rxmsg_checksum = 0;

```

```

        no_HOST_Rx_timer = 0;
        return;
    }

case RECEIVING_MESSAGE_DATA:
    no_HOST_Rx_timer = 255;                      // Set Rx timeout to 255
msecs

/* Now see if we are at the end of the message buffer */
if (HOST_Rxmsg_byte_cnt < (MAX_GDFS_RX_DATA +
sizeof(recv_msg_header)))
{
    *HOST_Rxmsg_byte_ptr++ = new_char;
    HOST_Rxmsg_checksum ^= new_char;
    HOST_Rxmsg_byte_cnt++;

/* Now see if we are at the end of the message */
if (HOST_Rxmsg_byte_cnt >= sizeof(recv_msg_header))
{
    msg_count = data_length + sizeof(recv_msg_header);
    if (msg_count == HOST_Rxmsg_byte_cnt)
    {
        HOST_Rxmsg_buf.msg_length = msg_count;
        HOST_Rxmsg_status = RECEIVING_CHECKSUM;
        HOST_Rxmsg_byte_ptr =
&HOST_Rxmsg_buf.checksum;
    }
    else if (msg_count < HOST_Rxmsg_byte_cnt)
    {
        HOST_Rxmsg_byte_ptr =
HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
        HOST_Rxmsg_byte_cnt = 0;
        HOST_Rxmsg_checksum = 0;
        no_HOST_Rx_timer = 0;
    }
}
else
{
    // Host just sent us a packet bigger than our buffer
    HOST_Rxmsg_byte_ptr =
&HOST_Rxmsg_buf.Rxmsg_header.start_char;
    HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
    HOST_Rxmsg_byte_cnt = 0;
    HOST_Rxmsg_checksum = 0;
    no_HOST_Rx_timer = 0;
}
break;
case RECEIVING_CHECKSUM:
    *HOST_Rxmsg_byte_ptr = new_char;
    /* Now see if the checksum matches */
    HOST_Rxmsg_checksum = 0x80 | (0x0F & HOST_Rxmsg_checksum);
    if (!(HOST_Rxmsg_checksum ^ HOST_Rxmsg_buf.checksum))
        HOST_Rxmsg_buf.in_use = TRUE;
    else
        flag_error(BAD_HOST_RX_MSG_CHECKSUM);
    HOST_Rxmsg_byte_ptr = &HOST_Rxmsg_buf.Rxmsg_header.start_char;
    HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
    HOST_Rxmsg_byte_cnt = 0;
    HOST_Rxmsg_checksum = 0;
    no_HOST_Rx_timer = 0;
    break;
case RECEIVING_EPROM_CMD:
    HOST_Rxmsg_status = RECEIVING_EPROM_DATA;
    no_HOST_Rx_timer = 255;                      // Set Rx timeout to 255 msecs
    switch (new_char)
    {
        case 0x55:      // Erase command
        case 0x66:      // Record command
        case 0x77:      // End command

```

```

        HOST_Rxmsg_status = RECEIVING_EPROM_DATA;
        no_HOST_Rx_timer = 255;                                // Set Rx timeout to 255
msecs

        *HOST_Rxmsg_byte_ptr++ = new_char; // Store byte
        HOST_Rxmsg_byte_cnt++;          // increment byte counter
        HOST_Rxmsg_checksum += new_char; // XOR with checksum
        break;
    default: // If not valid command, start over
        HOST_Rxmsg_byte_ptr = &HOST_Rxmsg_buf.Rxmsg_header.start_char;
        HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
        HOST_Rxmsg_byte_cnt = 0;
        HOST_Rxmsg_checksum = 0;
        no_HOST_Rx_timer = 0;
        return;
    }
    break;
case RECEIVING_EPROM_DATA:
    no_HOST_Rx_timer = 255;                                // Set Rx timeout to 255
msecs

/* Now see if we are at the end of the message buffer */
if (HOST_Rxmsg_byte_cnt < (MAX_GDFS_RX_DATA))
{
    *HOST_Rxmsg_byte_ptr++ = new_char;
    HOST_Rxmsg_checksum += new_char;
    HOST_Rxmsg_byte_cnt++;
// S0BUF=HOST_Rxmsg_byte_cnt; // TEST ONLY ****
*****msg_count = data_length;
// S0BUF=new_char; // TEST ONLY ****
if ((msg_count-1) ==
HOST_Rxmsg_byte_cnt)&&(new_char==0x03)
{
    HOST_Rxmsg_buf.msg_length = msg_count;
    HOST_Rxmsg_status =
RECEIVING_EPROM_CHECKSUM;
    HOST_Rxmsg_byte_ptr =
&HOST_Rxmsg_buf.checksum;
}
else //if (msg_count < HOST_Rxmsg_byte_cnt)
{
    S0BUF=new_char; // TEST ONLY ****
    HOST_Rxmsg_byte_ptr =
&HOST_Rxmsg_buf.Rxmsg_header.start_char;
    HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
    HOST_Rxmsg_byte_cnt = 0;
    HOST_Rxmsg_checksum = 0;
    no_HOST_Rx_timer = 0;
}
}
else
{
    // Host just sent us a packet bigger than our buffer
    HOST_Rxmsg_byte_ptr =
&HOST_Rxmsg_buf.Rxmsg_header.start_char;
    HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
    HOST_Rxmsg_byte_cnt = 0;
    HOST_Rxmsg_checksum = 0;
    no_HOST_Rx_timer = 0;
}
break;
case RECEIVING_EPROM_CHECKSUM:
    *HOST_Rxmsg_byte_ptr = new_char;
// S0BUF=new_char; // TEST ONLY ****
/* Now see if the checksum matches */
if (!(HOST_Rxmsg_checksum ^ HOST_Rxmsg_buf.checksum))

```

```

        {
            HOST_Rxmsg_buf.in_use = TRUE;
        }
        else
            flag_error(BAD_HOST_RX_MSG_CHECKSUM);
        HOST_Rxmsg_byte_ptr = &HOST_Rxmsg_buf.Rxmsg_header.start_char;
        HOST_Rxmsg_status = WAITING_FOR_START_CHAR;
        HOST_Rxmsg_byte_cnt = 0;
        HOST_Rxmsg_checksum = 0;
        no_HOST_Rx_timer = 0;
        break;

    default:
        break;
    }
}

/*
 * Has a character just been transmitted to GDFS?
 */
if ( TIO )
{
    TIO = 0;                                // Clear interrupt pending condition
    sending_ACK = FALSE;                     // Clear this bit just in case
/*
 * Is there anything in the host transmit buffer?
 */
switch (HOST_Txmsg_status)
{
    case SENDING_DATA:
        if (HOST_Txmsg_byte_cnt)
        {
            /* Send next character in transmit buffer out the HOST UART */
            HOST_Txmsg_checksum ^= *HOST_Txmsg_byte_ptr;
            S0BUF = *HOST_Txmsg_byte_ptr++;
            HOST_Txmsg_byte_cnt--;
        }
        else
        {
            HOST_Txmsg_status = SENDING_CHECKSUM;
            HOST_Txmsg_checksum = 0x80 | (HOST_Txmsg_checksum & 0x0F);
            S0BUF = HOST_Txmsg_checksum;
        }
        break;
    case SENDING_ACK:
        HOST_Txmsg_buf.in_use = FALSE;
        HOST_Txmsg_status = TXMSG_COMPLETE;
        break;
    case SENDING_CHECKSUM:
    default:
        HOST_Txmsg_buf.in_use = FALSE;
        HOST_Txmsg_status = TXMSG_COMPLETE;
        break;
    }
}
}

```

```

/*
* Name:          CAMERA Interrupt Service Routine (ISR)
*
* rf_isr();
*
* Description:
*
* This ISR sends and receives character to and from the Serial 1 Port
*
* Returns:
*
*/
void camera_isr( void ) interrupt 16 using 1 // 0x3B; ser port 1, camera
{
    unsigned char new_char;
    int data_length;          // length the message should be
    bit ignore_first_50;

    /*
     * Has a character been received?
     */
    if ( S1CON & 0x01 )
    {
        //////////////////////////////////////////////////
        // test_mode(S1BUF); //*****DELETE THIS *****
        //////////////////////////////////////////////////
        S1CON &= 0xFE;           // Clear interrupt pending condition

        new_char = S1BUF;

        //////////////////////////////////////////////////
        // test_mode('C'); //*****DELETE THIS *****
        //////////////////////////////////////////////////
        switch (CAMERA_Rxmsg_status)
        {
            case WAITING_FOR_START_CHAR:
                if ( !(new_char ^ START_CHAR) )      // If a start char
                {
                    CAMERA_Rxmsg_status = RECEIVING_ID;
                    CAMERA_Rxmsg_byte_ptr = &CAMERA_Rxmsg_buf.Rxmsg_header.start_char;
                    CAMERA_Rxmsg_byte_cnt = 1;
                    CAMERA_Rxmsg_checksum = new_char;
                    no_CAMERA_Rx_timer = 255;           // Set Rx timeout to 255 msecs
                    *CAMERA_Rxmsg_byte_ptr++ = new_char; // Store byte
                    break;
                }
                if (new_char == ACK)
                    CAMERA_ACK_timer = 0; // If ACK, disable ACK timer
                if (new_char == NACK)
                    CAMERA_ACK_timeout = TRUE; // If NACK, flag ACK failure
                if (new_char ^ START_CHAR) // If not a valid char
                {
                    HOST_Rxmsg_status=TEST_MODE;
                    S0BUF = new_char;      // just reflect it to host.
                    return;
                }
                break;
            case RECEIVING_ID:
                if (new_char != ID)    // If not ID, start over
                {
                    CAMERA_Rxmsg_byte_ptr = &CAMERA_Rxmsg_buf.Rxmsg_header.start_char;
                    CAMERA_Rxmsg_status = WAITING_FOR_START_CHAR;
                    CAMERA_Rxmsg_byte_cnt = 0;
                    CAMERA_Rxmsg_checksum = 0;
                    no_CAMERA_Rx_timer = 0;
                    break;
                }
                CAMERA_Rxmsg_status = RECEIVING_TYPE;
                no_CAMERA_Rx_timer = 255;           // Set Rx timeout to 255 msecs
        }
    }
}

```

```

*CAMERA_Rxmsg_byte_ptr++ = new_char; // Store byte
CAMERA_Rxmsg_byte_cnt++; // increment byte counter
CAMERA_Rxmsg_checksum ^= new_char; // XOR with checksum
break;
case RECEIVING_TYPE: // Stored in "data_length" slot
    if (new_char != 'c') // If not ID, start over
    {
        CAMERA_Rxmsg_byte_ptr = &CAMERA_Rxmsg_buf.Rxmsg_header.start_char;
        CAMERA_Rxmsg_status = WAITING_FOR_START_CHAR;
        CAMERA_Rxmsg_byte_cnt = 0;
        CAMERA_Rxmsg_checksum = 0;
        no_CAMERA_Rx_timer = 0;
        break;
    }
    CAMERA_Rxmsg_status = RECEIVING_CMD;
    no_CAMERA_Rx_timer = 255; // Set Rx timeout to 255 msecs
    *CAMERA_Rxmsg_byte_ptr++ = new_char; // Store byte
    CAMERA_Rxmsg_byte_cnt++; // increment byte counter
    CAMERA_Rxmsg_checksum ^= new_char; // XOR with checksum
    ignore_first_50 = FALSE; // Clear this flag
    break;
case RECEIVING_CMD:
    switch (new_char)
    {
        case 'C':
            ignore_first_50 = TRUE; // Set flag to ignore 50 bytes
        default:
            data_length = 25;
            CAMERA_Rxmsg_status = RECEIVING_MESSAGE_DATA;
            no_CAMERA_Rx_timer = 255; // Set Rx timeout to 255
msecs
            *CAMERA_Rxmsg_byte_ptr++ = new_char; // Store byte
            CAMERA_Rxmsg_byte_cnt++; // increment byte counter
            CAMERA_Rxmsg_checksum ^= new_char; // XOR with checksum
            break;
    }
case RECEIVING_MESSAGE_DATA:
    no_CAMERA_Rx_timer = 30; // Set Rx timeout to
30 msecs
    if ((ignore_first_50 == TRUE) && (CAMERA_Rxmsg_byte_cnt < 50))
    {
        CAMERA_Rxmsg_byte_cnt++; // increment byte counter
        CAMERA_Rxmsg_checksum ^= new_char; // XOR with checksum
        break;
    }
    if ((ignore_first_50 == TRUE) && (CAMERA_Rxmsg_byte_cnt >= 50))
    {
        CAMERA_Rxmsg_byte_cnt = 7; // Reset byte counter to 7 &
        ignore_first_50 = FALSE; // clear flag
    }
/* Now see if we are at the end of the message buffer */
    if (CAMERA_Rxmsg_byte_cnt >= 32)
    {
        CAMERA_Rxmsg_byte_ptr = &CAMERA_Rxmsg_buf.Rxmsg_header.start_char;
        CAMERA_Rxmsg_status = WAITING_FOR_START_CHAR;
        CAMERA_Rxmsg_byte_cnt = 0;
        CAMERA_Rxmsg_checksum = 0;
        no_CAMERA_Rx_timer = 0;
        break;
    }
    if (new_char == ETX) // End of message
    {
        CAMERA_Rxmsg_checksum ^= *CAMERA_Rxmsg_byte_ptr; // Undo prev. XOR
        /* Now see if the checksum matches */
        CAMERA_Rxmsg_checksum = 0x80 | (0x0F & CAMERA_Rxmsg_checksum);
        if (!(CAMERA_Rxmsg_checksum ^ *CAMERA_Rxmsg_byte_ptr))
            CAMERA_Rxmsg_buf.in_use = TRUE;
        else
            flag_error(BAD_CAMERA_RX_MSG_CHECKSUM);
        CAMERA_Rxmsg_byte_ptr = &CAMERA_Rxmsg_buf.Rxmsg_header.start_char;
    }
}

```

```

        CAMERA_Rxmsg_Status = WAITING_FOR_START_CHAR;
        CAMERA_Rxmsg_BytCntr = 0;
        CAMERA_Rxmsg_Checksum = 0;
        no_CAMERA_Rx_Timer = 0;
        break;
    }
    no_CAMERA_Rx_Timer = 255; // Set Rx timeout to 255 msecs
    *CAMERA_Rxmsg_BytPtr++ = new_char; // Store byte
    CAMERA_Rxmsg_BytCntr++; // increment byte counter
    CAMERA_Rxmsg_Checksum ^= new_char; // XOR with checksum
    break;
}
/*
 * Has a character just been transmitted to the Camera?
 */
if ( S1CON & 0x02 )
{
    S1CON &= 0xFD; // Clear interrupt pending condition
// test_mode('c'); //*****DELETE THIS*****
}

switch (CAMERA_Txmsg_Status)
{
    case SENDING_DATA:
        if (CAMERA_Txmsg_BytCntr)
        {
            /* Send next character in transmit buffer out the camera UART */
            CAMERA_Txmsg_Checksum ^= *CAMERA_Txmsg_BytPtr;
            S1BUF = *CAMERA_Txmsg_BytPtr++;
            CAMERA_Txmsg_BytCntr--;
        }
        else
        {
            CAMERA_Txmsg_Status = SENDING_CHECKSUM;
            CAMERA_Txmsg_Checksum = 0x80 | (CAMERA_Txmsg_Checksum & 0x0F);
            S1BUF = CAMERA_Txmsg_Checksum;
        }
        break;
    case SENDING_CHECKSUM:
        CAMERA_Txmsg_Status = SENDING_END_OF_MSG;
        S1BUF = 0x03;
        break;
    case SENDING_END_OF_MSG:
    default:
        CAMERA_Txmsg_Buf.in_use = FALSE;
        CAMERA_Txmsg_Status = TXMSG_COMPLETE;

        /* Enable the receiver in the UART (Don't know why) */
        S1CON |= 0xB0;
        break;
}
}

```

```

*****
* Name:          Ext 1 Interrupt Service Routine (ISR)
*
*           Msec_tick_timer();
*
* Description:
*
*   This ISR will be executed when the Internal Timer 0 occurs
*
* Returns:       nothing
*
*****
void Msec_tick_timer( void ) interrupt 1 using 2      // 0B; Timer 0
{
    // Reload timer with 1 msec timeout value.
    // Should be FBC3.
    TH0 = 0xFB;
    TL0 = 0xC3;

    if (CAMERA_ACK_timer) // If waiting for ACK, decrement timer & check
    {
        if (--CAMERA_ACK_timer == 0)
        {
            CAMERA_ACK_timeout = TRUE;
        }
    }
    if (HOST_ACK_timer)    // If waiting for ACK, decrement timer & check
    {
        if (--HOST_ACK_timer == 0)
            HOST_ACK_timeout = TRUE;
    }
    if (T0_msec_counter)
    {
        if (--T0_msec_counter == 0)
            wait_for_T0_timeout = FALSE;
    }
    if (no_HOST_Rx_timer)
    {
        if (--no_HOST_Rx_timer == 0)
            HOST_Rx_timeout = TRUE;
    }
    if (no_CAMERA_Rx_timer)
    {
        if (--no_CAMERA_Rx_timer == 0)
            CAMERA_Rx_timeout = TRUE;
    }
    if (digital_zoom_in_timer)
    {
        digital_zoom_in_timer--;
        if (digital_zoom_in_timer==0) zoom_stop();
        zoom_estimate++;
        if (zoom_estimate>2000) zoom_estimate=2000;
    }
    if (digital_zoom_out_timer)
    {
        digital_zoom_out_timer--;
        if (digital_zoom_out_timer==0) zoom_stop();
        zoom_estimate--;
        if (zoom_estimate<0) zoom_estimate=0;
    }

// This section reads the zoom and focus values from the A/D converter and stores
// them in zoom_status or focus_status.  If the value read is higher or lower, the
// *_status is incremented or decremented by one.  This is to prevent noise on the
// potentiometers from causing wild swings in *_status values.  I am only using the
// upper 8 bits of the A/D (ADDATH).

    if (checking_zoom)
    {
        if (ADDATH > zoom_status) zoom_status++; // Read & filter zoom from A/D
        if (ADDATH < zoom_status) zoom_status--;
    }
}

```

```

        checking_zoom = FALSE; // Clear flag so focus will be checked next time
        ADCON1 = 0x81;           // Set MUX to "focus" input & /8 clock
        ADDATL = 0;              // Start conversion
    }
    else
    {
        if (ADDATH > focus_status) focus_status++; // Read & filter focus
        if (ADDATH < focus_status) focus_status--;
        checking_zoom = TRUE; // Set flag so zoom will be checked next time
        ADCON1 = 0x80;           // Set MUX to "zoom" input & /8 clock
        ADDATL = 0;              // Start conversion
    }

// This section checks for lack of progress in zoom or focus. If the values aren't
// changing, the motor may have hit the stop, so the *_AD setting is changed to
// the value read. Value of A/D is checked every ~256 ms. If 2 checks are identical
// it is assumed the stop is reached. I estimate that the value should change about
// once every 16 ms.

if (P5==0x27 || P5==0x3B)
{
    if (zooming == 0)      // If zooming just started, start zoom timing
    {
        zooming=255;
        old_zoom_status=zoom_status; // and save status
    }
    else if (--zooming==0) // If timer just ran out, check for progress
    {
        if (!(zoom_status ^ old_zoom_status))
        {
            zoom_AD=zoom_status; // If no progress, stop zoom
            P5=0x2B & (P5 | 0xFE);
            zooming=0;
        }
        else
        {
            old_zoom_status=zoom_status; // If progress, reset timer
            zooming=255;
        }
    }
    else zooming=0;
}
if (P5==0x0F || P5==0x6B)
{
    if (focusing == 0)      // If focusing just started, start focus timing
    {
        focusing=255;
        old_focus_status=focus_status; // and save status
    }
    else if (--focusing==0) // If timer just ran out, check for progress
    {
        if (!(focus_status ^ old_focus_status))
        {
            focus_AD=focus_status; // If no progress, stop focus
            P5=0x2B & (P5 | 0xFE);
            focusing=0;
        }
        else
        {
            old_focus_status=focus_status; // If progress, reset timer
            focusing=255;
        }
    }
}
else focusing=0;
}

```

```

/*
 * RIC.H
 *
 * Include file for VIS for hardware/software/system defines
 *
 * History: Adapted for VIS 12/2/98
 *
 ****
 */

#include <REG517A.H>           /* Microcontroller specific registers */
#include <STRING.H>
#include <ABSACC.H>
#include <STDIO.H>

#define SOFTWARE_VER 0x01      // This version of Boot Software
#define HARDWARE_VER 0x00      // Version of hardware/Database

#define download_status     DBYTE[0xFF]    // Keeps track of RF download reply
#define boot_SW_version    DBYTE[0xFE]    // Pass on the boot software version

sbit A16_NOT      = P3^4;    // Bit 4 is A16 inverted
sbit RUN_IN_FLASH = P3^5;    // Bit 5 controls running in Flash/RAM
#define FLASH_A16LOW 0x0FF // default Port 3 output; runs out of low Flash
#define RAM_A16HI   0x0CF // used when loading application into Flash
#define RAM_A16LOW  0x0DF // used when replacing boot code in Flash
#define FLASH_A16HI 0x0EF // used when executing application code
#define A16_bit      4        // used for bitwise applications
#define RAM_bit      5        // "
****

* CONSTANTS
****

typedef signed char sbyte;
typedef bit      bool;
typedef unsigned int word;
typedef unsigned long ulong;

#define TRUE      1
#define FALSE     0
#define ON       1
#define OFF      0

#define START_CHAR 0xF8      // Packet start character
#define ID        0x01      // VIS ID character
#define ACK       0x06      // Acknowledge character
#define NACK      0x15      // Not-acknowledge character
#define ETX       0x03      // End-of-Transmission character
#define ADTOL     2         // A/D tolerance allowed (used in RIC)
                    // Set this tolerance to 2 for 6", 3 for 4" cameras
#define IRIS_DEFAULT 40      // Approx gain of zero for iris (APL)
#define IRIS_STEP    4       // Amount of step adjustment of iris
#define HOST_MSG     0       /* Message Rx/Tx status passed as input */
#define PASS        0x00      /* Pass condition of the self test */

// Flash Memory Registers and bit masks
#define DATA_REG0      XBYTE[0x5555]
#define DATA_REG1      XBYTE[0x2AAA]
#define DEVICE_REG     XBYTE[1]
#define MANUF_REG      XBYTE[0]
#define DATA_POLLING   0x80
#define TIMEOUT        0x20

// Flash Memory Commands
#define FLASH_RESET    0xF0
#define FLASH_ID       0x90
#define FLASH_PROGRAM  0xA0
#define ERASE_ENABLE   0x80
#define CHIP_ERASE    0x10
#define SECTOR_ERASE  0x30

```

```

#define FLASH_CHECKSUM      0xAAAA
#define CHECKSUM_SIZE       sizeof(int)
#define BOOT_CODE           0
#define APP_CODE            1
#define APP_SIZE            0x8000
#define BOOT_SIZE           0x4000
#define SEGMENT_SIZE        0x4000

// Startup Flag Masks
#define POWER_UP            0x00
#define ERASE_MASK          0x01
#define BEGIN_ERASE         0x02
#define INVALID_APP          0x04
#define APP_GOOD             0x08

/*
 * Message types
 */
#define SELF_TEST            1      /* do a self test */ */
#define ACK_MSG              2      /* acknowledge a message */ */
#define CTL_MSG              3      // Control message (No ACK)
#define SET_UP                4      /* set up variables */ */
#define SEND_MSG              6      /* send a message */ */
#define ERROR_MSG             13     /* An error has occurred (1 byte param) */
#define BEGIN_DOWNLOAD        0x55   // Initiate download from host to FLASH
#define INTEL_HEX_RECORD     0x66   // Programming record (cnt/addr/data)
#define END_DOWNLOAD          0x77   // Download from host to FLASH is done
#define REBOOT                0x88   // Let watchdog timeout

// Error Message Types
#define APP_BLOCK_BAD         0x31   // Application FLASH block cksum bad
#define FLASH_ERASE_ERROR     0x32   // Erase cycle on FLASH went bad
#define FLASH_PROGRAM_ERROR   0x33   // Programming cycle on FLASH went bad
#define HOST_RX_TIMEOUT_ERROR 0x34   // Msg from Host started, no end
#define RF_RX_TIMEOUT_ERROR   0x35   // Msg from Rx (radio) started, no end
#define BAD_HARDWARE_ERROR     7      // Bad A/D converter/hardware problems
#define NO_HOST_BUFFERS_ERROR 10     // All Rx buffers in use error
#define NO_RF_BUFFERS_ERROR   11     // All Tx/Rx buffers in use error
#define BAD_HOST_CHKSUM_ERROR 13     // Host checksum was bad
#define BAD_RF_CHKSUM_ERROR   14     // RF checksum was bad
#define MISC_CONTROLLER_ERROR  16     // Error condition (misc)
#define ILLEGAL_MSG_ERROR     18     // Msg doesn't match mode/unrecognized
#define HEX_RECORD_SEQ_ERROR  23     // Sequence # error in hex record
#define ERROR_4                44     // Just for testing

/* Internal Hardware/Software Timer and Timeout values */
#define TMR1_CNTRL_REG 0x20      /* GATE = 0, Counter operation, */
                                /* and Mode 2 (8bit auto-reload). */
#define TMR0_CNTRL_REG  0x01      /* GATE = 0, Timer operation, */
                                /* and Mode 1 (16 bit). */
#define TMR2_CNTRL_REG  0x30      /* 16 bit autoload for RF baud rate */

// Delay and timer values, wait loop implementation as a macro
#define WAIT(limit) \
{ \
    register char counter; \
    for ( counter = 0; counter < limit; counter++ ) \
    { \
    } \
}

#define DELAY_10_US           5      /* for WAIT macro */
#define DELAY_300_US          100    /* for WAIT macro */

// Message Lengths
#define SETUP_DATA_LEN         16    // Is 16 OK? (was sizeof(eeprom_type) + 5)
#define BEGIN_DOWNLOAD_LEN     1
#define ERROR_MSG_LEN          1

```

```

/* Transmit message states for transmit_rf/host_message_status */
typedef enum
{
    TXMSG_COMPLETE = 0,
    SENDING_DATA,
    SENDING_CHECKSUM,
    SENDING_END_OF_MSG,
    SENDING_ACK
} transmit_states;

typedef enum
{
    WAITING_FOR_START_CHAR = 0,
    RECEIVING_ID,
    RECEIVING_TYPE,
    RECEIVING_CMD,
    RECEIVING_MESSAGE_DATA,
    RECEIVING_CHECKSUM,
    RECEIVING_EPROM_CMD,
    RECEIVING_EPROM_DATA,
    RECEIVING_EPROM_CHECKSUM,
    TEST_MODE,
    TEST_MODE2
} receive_states;

#define MAX_GDFS_RX_DATA      48      // Max message size of Intel Hex Record +5
#define MAX_GDFS_TX_DATA      8       // Max message size for GDFS +2
#define MAX_CAMERA_RX_DATA    35      // Should be 75. Max message size from camera +1
#define MAX_CAMERA_TX_DATA    5       // Max message size to camera +2

// Description of header info for RF/Host messages
typedef struct header_info
{
    unsigned char start_char;          /* Destination ID this message being sent to */
    unsigned char id;                 /* Length of packet data following header */
    unsigned char data_length;         /* Message type being sent/received */
    unsigned char cmd;
} rcv_msg_header;

typedef struct GDFS_Rxmsg_buf          /* Rx message buf description */
{
    unsigned char in_use;             /* Indicates data present */
    int msg_length;                 /* Number of bytes in buffer */
    rcv_msg_header Rxmsg_header;     /* RF/Host message header */
    unsigned char msg_data[MAX_GDFS_RX_DATA]; /* the actual Rx data bytes */
    unsigned char checksum;           /* Chksum of bytes in buffer */
} GDFS_Rxmsg_buf;

typedef struct Cam_Rxmsg_buf          /* Rx message buf description */
{
    unsigned char in_use;             /* Indicates data present */
    int msg_length;                 /* Number of bytes in buffer */
    rcv_msg_header Rxmsg_header;     /* RF/Host message header */
    unsigned char msg_data[MAX_CAMERA_RX_DATA]; /* the actual Rx data bytes */
    unsigned char checksum;           /* Chksum of bytes in buffer */
} Cam_Rxmsg_buf;

typedef struct tx_header_info
{
    /* Length of packet data following header */
    /* Destination ID this message being sent to */
    /* Message type being sent/received */
} tx_msg_header;

typedef struct GDFS_Txmsg_buf          /* Tx message buf description */
{
    unsigned char in_use;             /* Indicates data present */
    int msg_length;                 /* Number of bytes in buffer */
    tx_msg_header Txmsg_header;      /* RF/Host message header */
}

```

```
    unsigned char msg_data[MAX_GDFS_TX_DATA]; /* the actual Tx data bytes */
} GDFS_Txmsg_buf;

typedef struct Cam_Txmsg_buf           /* Tx message buf description */
{
    unsigned char in_use;           /* Indicates data present */
    int msg_length;                /* Number of bytes in buffer */
    tx_msg_header Txmsg_header;    /* RF/Host message header */
    unsigned char msg_data[MAX_CAMERA_TX_DATA]; /* the actual Tx data bytes */
} Cam_Txmsg_buf;

typedef enum
{
    HOST_Rx_TIMEOUT,
    BAD_HOST_Rx_MSG_CHECKSUM,
    BAD_CAMERA_Rx_MSG_CHECKSUM,
    BAD_APPLICATION_BLOCK,
    BAD_FLASH_ERASE,
    BAD_FLASH_PROGRAMMING,
    BAD_OTHER_4
} error_types;
```

```
*****
* Name:      PROTO.H          (Prototypes for RIC.C)
*
*
* History: Adapted for VIS 12/2/98
*
*****
/*****
```

```
extern void main( void );
extern void init_globals( void );
extern void init_sys( void );
extern void calibrate( void );
extern void flag_error(unsigned char error_type) reentrant;
extern void process_camera_msg(void);
extern void set_iris(unsigned char iris);      // Set iris (send APL value to camera)
extern void send_latch_status(void);           // Send iris & power status
extern void start_camera_msg(void);
extern void send_ack(void);
extern void send_lens_status(void);            // Send lens status to GDFS
extern void test_mode(unsigned char flagbyte); // Special test mode
extern void request_lens_status(void); // Send lens status request to camera
extern void zoom_in(void);                   // Start digital zoom in
extern void zoom_out(void); . . .           // Start digital zoom out
extern void zoom_stop(void);                 // Stop digital zoom
extern void test_mode2(unsigned char flagbyte); // Special test mode
```

```
*****
* Prototypes in HOSTMSG.C
*****
/*****
```

```
extern void process_host_msg( void );
extern void process_camera_msg( void );
extern void start_host_msg(void);
```

```
*****
* Prototypes in FLASH.C
*****
/*****
```

```
extern void flash_cmd(unsigned char cmd);
extern unsigned char ascii_to_byte(unsigned char ascii_byte);
extern bit amd_program_buffer(unsigned int base_address, unsigned char idata *buf, unsigned char len);
extern bit program_buffer(unsigned char idata *buf);
extern bit program_checksum(void);
extern bit amd_erase(unsigned int sector_address);
extern bit erase_flash(unsigned char block);
extern bit check_flash(unsigned char block);
```

```
*****
* Functions in RICASM.A51
*****
/*****
```

```
extern void jump_to_app(void);
extern char jump_to_ram(void);
extern char ram_return(void);
extern void reboot(void);
extern void _return_to_boot(void);
// extern void testcode(void);
```

```

*****
* Name: RICGLOB.H
*
* Description:
*
* Contains external definitions for global variables.
*
* History: Adapted for VIS 12/2/98
*
*****
extern bit sending_ACK;                                // True when ACK is being sent to host
extern bit HOST_ACK_timeout;                          // Indicates Host ACK timer expired
extern bit CAMERA_ACK_timeout;                      // Indicates Camera ACK timer expired
extern bit HOST_Rx_timeout;                          // Timeout flag for GDFS Rx data stopping
extern bit CAMERA_Rx_timeout;                      // Timeout flag for camera Rx data stopping
extern bit sending_to_camera;                      // Host is sending to camera (started in APP)
extern bit wait_for_T0_timeout;                     // 1 msec tick generator
extern unsigned char startup_flags;                 // Used by flash programming functions
extern char prog_block;                            // Flash programming block
extern unsigned char P3_latch_image;                // Image of data written to Port 3
extern unsigned char no_HOST_Rx_timer;              // Timeout counter for Host Rx (bad rx msg)
extern unsigned char no_CAMERA_Rx_timer;            // Timeout counter for Camera Rx (bad
rx msg)
extern unsigned char T0_msec_counter;               // 1 msec tick timer/counter
extern int digital_zoom_in_timer;                  // 1 msec digital zoom timer/counter
extern int digital_zoom_out_timer;                 // 1 msec digital zoom timer/counter
extern unsigned char RF_Txmsg_retries;             // Counter keeping track of reply retries
extern unsigned char HOST_Rxmsg_status;            // State of the Rx data stream from Host
extern unsigned char HOST_Txmsg_status;            // State of the Tx data stream to Host
extern unsigned char HOST_Rxmsg_byte_cnt;          // Keeps track of the byte count
extern unsigned char HOST_Txmsg_byte_cnt;          // Keeps track of the byte count
extern unsigned char HOST_Rxmsg_checksum;          // Keeps a running tally of checksum
extern unsigned char HOST_Txmsg_checksum;          // Keeps a running tally of checksum

extern unsigned char CAMERA_Rxmsg_status;          // State of the Rx data stream from CAMERA
extern unsigned char CAMERA_Txmsg_status;          // State of the Tx data stream to CAMERA
extern unsigned char CAMERA_Rxmsg_byte_cnt;        // Keeps track of the byte count
extern unsigned char CAMERA_Txmsg_byte_cnt;        // Keeps track of the byte count
extern unsigned char CAMERA_Rxmsg_checksum;         // Keeps a running tally of checksum
extern unsigned char CAMERA_Txmsg_checksum;         // Keeps a running tally of checksum

extern int last_dnload_msg_state;                 // State of RF dnload msgs (ignore retries)
extern int HOST_ACK_timer;                        // Timer for Host ACK reply
extern int CAMERA_ACK_timer;                      // Timer for Camera ACK reply
extern unsigned char idata *HOST_Rxmsg_byte_ptr;   // Points to current position in Rxmsg_buf
extern unsigned char idata *HOST_Txmsg_byte_ptr;   // Points to current position in Txmsg_buf
extern GDFS_Rxmsg_buf idata HOST_Rxmsg_buf;       // Host Rxmsg buffer (data from host)
extern GDFS_Txmsg_buf idata HOST_Txmsg_buf;       // Host Txmsg buffer (data to host)

extern unsigned char idata *CAMERA_Rxmsg_byte_ptr; // Points to current position in Rxmsg_buf
extern unsigned char idata *CAMERA_Txmsg_byte_ptr; // Points to current position in Txmsg_buf
extern Cam_Rxmsg_buf idata CAMERA_Rxmsg_buf;      // CAMERA Rxmsg buffer (data from CAMERA)
extern Cam_Txmsg_buf idata CAMERA_Txmsg_buf;      // CAMERA Txmsg buffer (data to CAMERA)

extern unsigned char zoom_status;                 // Latest measured value of zoom
extern unsigned char focus_status;                // Latest measured value of focus
extern int zoom_cmd;                            // Decoded 12 bit GDFS zoom command
extern int focus_cmd;                           // Decoded 12 bit GDFS focus command
extern unsigned char latch_status;               // Bit 0 - Iris status. 1=manual, 0=automatic
                                                // Bit 1 - Camera Power. 1=on, 0=off
extern unsigned char iris_setting;              // Iris gain setting
extern unsigned char zoom_AD;                   // Value desired for zoom A/D converter
extern int zoom_digital;                        // Value desired for digital zoom
extern int zoom_estimate;                       // Calculated digital zoom value
extern unsigned char focus_AD;                  // Value desired for focus A/D converter
extern unsigned char zoom_min;                  // Measured minimum value the pot can reach
extern unsigned char zoom_max;                  // Measured maximum value the pot can reach
extern unsigned char focus_min;                 // Measured minimum value the pot can reach

```

```
extern unsigned char focus_max;           // Measured maximum value the pot can reach
extern bit zoom_in_process;              // True if digital zooming is in process

#define ack_timeout 130                  // Number of msecs of timeout for 9.6 K baud

// Transferred here from RICISR 1ms interrupt routine
extern bit checking_zoom;
extern unsigned char zooming;
extern unsigned char focusing;
extern unsigned char old_zoom_status;
extern unsigned char old_focus_status;
```

Assembly language source code:

```
;-----  
; Name: RICASM.A51  
;  
; Power-On Initialization of Internal Memory (XDATA in 'C' in RAM self test)  
;  
; History: Adapted for VIS 3/16/99  
;  
;  
; With the following EQU statements the initialization of memory  
; at processor reset can be defined:  
;  
DNLOAD_STATUS EQU 0FFH ; Dnload status after rebooting/power up  
VERSION_NUM EQU 0FEH ; Dnload status after rebooting/power up  
IDATATOP EQU 0FDH ; the top of IDATA memory cleared at startup  
;  
; Stack Space for reentrant functions in the SMALL model.  
IBPSTACK EQU 1 ; set to 1 if small reentrant is used.  
IBPSTACKTOP EQU 0FDH+1 ; set top of stack to highest location+1.  
;  
-----  
NAME ?C_STARTUP  
  
.CSEG AT 0  
?C_C51STARTUP SEGMENT CODE  
APP_ENTRY SEGMENT CODE  
BOOT_RETURN SEGMENT CODE  
?STACK SEGMENT IDATA  
  
RSEG ?STACK  
DS 1  
  
EXTRN CODE (?C_START)  
EXTRN DATA (?C_IBP)  
EXTRN DATA (startup_flags)  
EXTRN DATA (P3_latch_image)  
PUBLIC ?C_STARTUP, jump_to_app, _jump_to_ram, _ram_return, reboot, _return_to_boot  
  
?C_STARTUP:  
    LJMP power_up  
  
RSEG ?C_C51STARTUP  
ver: DB '(Software Version 1 Revision 0)'  
power_up:  
    MOV R0,#DNLOAD_STATUS  
    CLR A  
    MOV @R0,A  
    MOV B,#0  
  
STARTUP1:  
    ;  
    ; Copy boot code from Flash to RAM.  
    ; R2:R3 = Count of bytes to be copied  
    ; R0:R1 = Pointer to destination RAM block  
    ; DPH:DPL = Pointer to source Flash block  
    ;  
    MOV P2,#0  
    MOV R0,#0  
    MOV R1,#0  
    MOV DPTR,#0  
    MOV R2,#40H  
    MOV R3,#00  
    CLR TR0 ; turn off timer 0  
  
    ;  
    ; Copy R2:R3 bytes to RAM.  
    ;  
copy_code_loop:  
    CLR A
```

```

MOVC A,@A + DPTR
INC DPTR
MOVX @R1,A
INC R1
DJNZ R3,copy_code_loop
INC R0
MOV P2,R0
DJNZ R2,copy_code_loop
MOV TH0,R6
MOV TL0,R7
c_preentry:
;
; Clear all of internal RAM.
;
MOV R0,#IDATATOP
CLR A
clear_loop:
MOV @R0,A
DJNZ R0,clear_loop

;
; Initialize the stack pointer and save the STARTUP_FLAGS.
;
MOV SP,#?STACK-1
MOV ?C_IBP,#IBPSTACKTOP
MOV startup_flags,B

;
; Start the C code and be running out of RAM
;
MOV P3_latch_image,#0DFH      ; save P3 setting (just in case)
MOV P3,#0DFH                  ; set P3 bits for low RAM memory

NOP
NOP

LJMP ?C_START

;
; JUMP_TO_APP jumps to the application program in upper Flash memory.
;
jump_to_app:
CLR EA
LJMP app_gateway

;
; _JUMP_TO_RAM jumps to the boot program in RAM memory, passing the flags
; passed to this function in R7, in B. Note that this code is position
; independent (it need not be aligned with anything). Jumping to RAM is just a
; transition from Boot code in Flash memory to RAM. The addresses are exactly
; the same.
;
jump_to_ram:
CLR EA
MOV P3_latch_image,#0DFH      ; save P3 setting (just in case)
MOV P3,#0DFH                  ; set P3 bits for low RAM memory
NOP
NOP

;
; Save the flags passed (via R7) in B.
;
MOV B,R7
LJMP c_preentry
;
; _RAM_RETURN jumps to the boot program in Flash memory, passing the flags
; passed to this function in R7, in B. Note that this code is position
; independent (it need not be aligned with anything). Returning to Flash is
; just a transition from boot code in RAM to boot code in Flash. The address
; ranges are exactly the same.
;

```

```

_ram_return:
    CLR EA
    MOV P3_latch_image,#0FFH ; save P3 setting (just in case)
    MOV P3,#0FFH ; set P3 bits for low Flash memory
    NOP
    NOP

    ;
    ; Save the flags passed (via R7) in B.
    ;
    MOV B,R7
    LJMP c_preentry

    ;
    ; APP_ENTRY/APP_GATEWAY is the gateway through which the boot code enters
    ; (jumps to) the application code. This segment is aligned with a
    ; corresponding segment in the application code so that setting the FA16
    ; bit and simultaneously clearing the RAM bit initiates execution from
    ; upper Flash memory (P3_latch_image will be initialized in app code)
    ;
RSEG APP_ENTRY
app_gateway:
    MOV P3,#0EFH ; set P3 bits for high Flash memory
    NOP
    NOP
STARTUP2:
    MOV R0,#IDATATOP
    CLR A
    MOV B,startup_flags
IDATALOOP:
    MOV @R0,A
    DJNZ R0, IDATALOOP

    MOV ?C_IBP,#IBPSTACKTOP
    MOV SP,#?STACK-1
    MOV startup_flags,B
    LJMP ?C_START

    ;
    ; BOOT_RETURN is the gateway through which application code can return to the
    ; boot block. This segment is aligned with a corresponding segment in the
    ; application code so that the first instruction after clearing FA16, app code
    ; transfers control to the boot code in Flash memory.
    ;
RSEG BOOT_RETURN
_return_to_boot:
    MOV P3,#0FFH ; set P3 bits for low Flash memory

    NOP
    NOP

    ;
    ; Save the startup flags passed from the application in B.
    ;
    MOV B,startup_flags
    LJMP STARTUP1

    ;
    ; REBOOT is a "function" which allows the C code to restart the system. Upon
    ; entry into this routine, it is assumed we are running in RAM. This routine
    ; jumps to the power up vector after it clears FA16 and FLS_PRG where
    ; execution continues from low Flash memory (the boot code).
    ;
reboot:
    MOV P3,#0FFH ; set P3 bits for low Flash memory
    NOP
    NOP
    LJMP power_up

    ;
    ; This is the interrupt 6 vector for use when the application code fails and
    ; a return to the boot code is required for chip reprogramming. It is
    ; accessed by momentarily grounding pin 6 of the microprocessor.
    ;

;invalid_app_reboot:

```

```
;      MOV B,#04H          ; Set startup_flags to indicate invalid app
;      LJMP _jump_to_ram    ; This is for downloading if app code fails
;
;      CSEG    AT      6BH    ; Interrupt 6 vector
;interrupt6:
;      LJMP invalid_app_reboot
;
END
```

```

# Name:      VISBOOT.MAK
#
# History:   Created in original VIS Firmware Design
#             (11/25/98) Version 1.0 is the first version (JLC)

AFLAGS      = DEBUG NOPRINT
CFLAGS      = SMALL MODDP2 CODE DEBUG OE SYMBOLS NOPRINT NOAREGS NOAMAKE
ASSEMBLER   = A51
COMPILER    = C51

# -----
# An implicit rule to build .OBJ files
# -----
visboot.hex : ric.obj ricisr.obj hostmsg.obj flash.obj ricasm.obj
    bl51 @visboot.lnk
    oh51 visboot.omf

.c.obj:
    $(COMPILER) $*.c $(CFLAGS)

.a51.obj:
    $(ASSEMBLER) $*.a51 $(AFLAGS)

# -----
# Targets and Dependencies
# -----
ric.obj:      ric.c ric.h proto.h visboot.mak
ricisr.obj:   ricisr.c ricglob.h ric.h proto.h visboot.mak
hostmsg.obj:  hostmsg.c ricglob.h ric.h proto.h visboot.mak
flash.obj:    flash.c ricglob.h ric.h proto.h visboot.mak
ricasm.obj:   ricasm.a51 visboot.mak

Utility file:
\vis\make\make -f visboot.mak
}

```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE January 2000	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE VIDEO IMAGING SYSTEM (VIS) INTERFACE BOARD FOR THE MOBILE INSHORE UNDERSEA WARFARE (MIUW) SYSTEM		5. FUNDING NUMBERS PE: OPN AN: DN301109 WU: MM48	
6. AUTHOR(S) J. Coleman, W. Marsh		8. PERFORMING ORGANIZATION REPORT NUMBER TD 3098	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SSC San Diego San Diego, CA 92152-5001		9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Space and Naval Warfare Systems Command PMW-182M San Diego, CA 92110-3127	
10. SPONSORING/MONITORING AGENCY REPORT NUMBER			
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This document describes the Video Imaging System (VIS) interface board that was designed by SSC San Diego and deployed in Mobile Inshore Undersea Warfare (MIUW) cameras.			
14. SUBJECT TERMS Mission Area: Surveillance Mobile Inshore Undersea Warfare (MIUW) video camera interface			15. NUMBER OF PAGES 74
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAME AS REPORT

21a. NAME OF RESPONSIBLE INDIVIDUAL J. Coleman	21b. TELEPHONE <i>(Include Area Code)</i> (619) 553-1266 e-mail: coleman@spawar.navy.mil	21c. OFFICE SYMBOL Code D372

INITIAL DISTRIBUTION

D0012	Patent Counsel	(1)
D0271	Archive/Stock	(6)
D0274	Library	(2)
D027	M. E. Cathcart	(1)
D0271	D. Richter	(1)
D372	J. L. Coleman	(15)

Defense Technical Information Center
Fort Belvoir, VA 22060-6218 (4)

SSC San Diego Liaison Office
Arlington, VA 22202-4804

Center for Naval Analyses
Alexandria, VA 22302-0268

Navy Acquisition, Research and
Development Information Center
Arlington, VA 22202-3734

Government-Industry Data Exchange Program
Operations Center
Corona, CA 91718-8000